Advanced Models and Programs

Scheme and Program Generation

Peter Sestoft

Mon 2009-02-23*

Outline

- Program generation
  - Programs that generate programs
- The Scheme programming language
  - Dynamically typed functional language
  - Concrete syntax = abstract syntax
- Closures in C# and Java
- Program generation in Scheme
  - Two-level languages
  - Distinguishing binding-times
- Runtime bytecode generation in C#/.NET

The power(n, x) function

- Computing x^n efficiently in Java/C#
- Using that $x^{2m} = (x^2)^m$ and $x^{m+1} = x*x^m$

```
public static int Power(int n, int x) {
  int p;
  p = 1;
  while (n > 0) {
    if (n % 2 == 0)
      { x = x * x; n = n / 2; }
    else
      { p = p * x; n = n - 1; }
  }
  return p;
}
```


Specialized power(n,x) for n=5

- What if we must compute x^5 for many x
- Then this function would be faster:
- (Why?)

```
public int Power_5(int x) {
  int p;
  p = 1;
  p = p * x;
  x = x * x;
  x = x * x;
  p = p * x;
  return p;
}
```

Generator of specialized power(n,x) functions

```
public static void PowerTextGen(int n) {
  System.out.println("public int Power_" + n + "(int x) {")
  System.out.println("  int p;");
  System.out.println("  p = 1;");
  while (n > 0) {
    if (n % 2 == 0) {
      System.out.println("  x = x * x;");
      n = n / 2;
    } else {
      System.out.println("  p = p * x;");
      n = n - 1;
    }
  }
  System.out.println("  return p;");
  System.out.println("}");
}
```


Binding times in Power(n,x)

- green=static=early, red=dynamic=late

```
public static int Power(int n, int x) {
  int p;
  p = 1;
  while (n > 0) {
    if (n % 2 == 0)
      { x = x * x; n = n / 2; }
    else
      { p = p * x; n = n - 1; }
  }
  return p;
}
```

- The generator performs the green code and prints (emits) the red code

The Scheme language

- Design by Guy L Steele 1978
  - Plus a revolutionary compilation technique
  - In his Master's thesis from MIT
  - Co-author of *Java Language Specification*
  - Now designing a new language "Fortress"
- Descends from Lisp (McCarthy 1960)
- A higher-order functional language
- Like Standard ML but no static types
- Simple syntax inherited from Lisp

Scheme expressions

- Compute 7+9:

```
(+ 7 9)
```

- Compute 7*9+13:

```
(+ (* 7 9) 13)
```

Prefix notation

- Define variable x to be 42:

```
(define x 42)
```

- Define variable x to be value of 7+9:

```
(define x (+ 7 9))
```

- If x<15 then x^2 else x-15:

```
(if (< x 15) (* x x) (- x 15))
```

Scheme function definitions

- Defining the function f(x) = x*3+7:

```
(define (f x) (+ (* x 3) 7))
```

- Same in C/C++/Java/C#:

```
int f(int x) { return x*3+7; }
```

- Calling the function on argument 10:

```
(f 10)
```

- Defining a recursive function:

```
(define (fac n)
  (if (= n 0)
      1
      (* n (fac (- n 1)))
) )
```

The power function in Scheme

```
(define (sqr x) (* x x))
(define (power n x)
  (if (> n 0)
      (if (eq? (remainder n 2) 0)
          (sqr (power (/ n 2) x))
          (* x (power (- n 1) x))
          )
      1)
  )

> (power 10 2)
1024
> (power 97 2)
158456325028528675187087900672
```

Scheme anonymous functions

- The anonymous function x → x*3+7

```
(lambda (x) (+ (* x 3) 7))
```

- Applying the function to argument 10:

```
((lambda (x) (+ (* x 3) 7)) 10)
```

- Anonymous functions in other languages

```
fn x => x*3+7
```

Standard ML, 1978

```
delegate(int x) { return x*3+7; }
```

C# 2.0

```
x => x*3+7
```

C# 3.0

```
λx . x*3+7
```

Lambda calculus, 1936

Closures

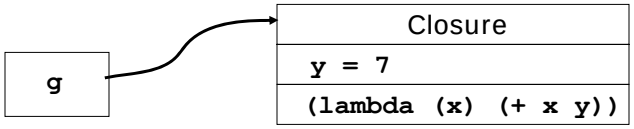
- An anonymous function may use a variable from an enclosing scope

```
(define (makeadd y)
  (lambda (x) (+ x y))
)
```

- A closure must be built for the function:

```
(define g (makeadd 7))
(g 42)
```

g's value is a closure



Scheme data: lists and pairs

- Scheme data are either
  - atoms (numbers, Booleans, symbols …) or
  - S-expressions: pairs, lists
- The list containing 11, 22 and 33:

```
'(11 22 33)
```

- Defining xs to be that list:

```
(define xs '(11 22 33))
```

- The first element of xs:

```
(car xs)
```

- The rest of xs:

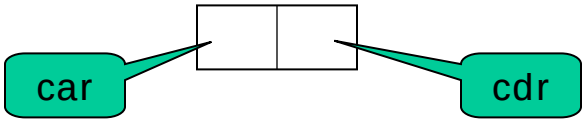
```
(cdr xs)
```

- The second element of xs:

```
(car (cdr xs))
```


Pairs and lists: s-expressions

- Structured data are built from cons cells
- A cons cell's components are car and cdr:



- Creating a new cons cell:

```
(cons 44 (+ 44 11))
```

44	55

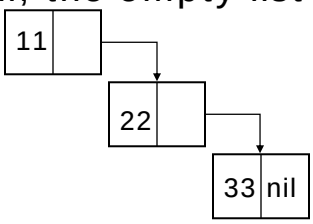
- A constant cons cell:

```
'(44 . 55)
```

44	55

A list is a special s-expression

- A list (11 22 33) is a right-linear tree ending in nil, the empty list ():



- Four ways to build the list:

```
'(11 22 33)
'(11 . (22 . (33 . ())))
(list 11 22 33)
(cons 11 (cons 22 (cons 33 ())))
```


Ten-minute exercise

- Assume **`xs`** is the list **`(11 22 33)`**
- Write Scheme expressions
  - for extracting the third element from **`xs`**
  - for extracting the list containing the third element from **`xs`**
  - for computing the sum of the first and second element of **`xs`**
  - for testing whether first element of **`xs`** is positive
- Write Scheme expression for 11+x*(22+x*(33+x*0))

Some list-processing functions

- Length of a list:

```
(define (len xs)
  (if (null? xs)
      0
      (+ 1 (len (cdr xs)))
))
```

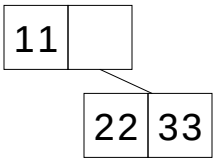
- Sum of a list's elements:

```
(define (sum xs)
  (if (null? xs)
      0
      (+ (car xs) (sum (cdr xs)))))
```

Some tree-processing functions

- Representing a tree:

```
(define t
  '(11 . (22 . 33))
)
```



- Depth of a tree:

```
(define (depth t)
  (if (pair? t)
      (+ 1 (max (depth (car t))
                (depth (cdr t))))
      0
))
```

Higher-order functions

- Mapping a function over a list:

```
(define (map f xs)
 (if (null? xs)
   ()
   (cons (f (car xs)) (map f (cdr xs)))
))
```

- Example use:

```
(define xs '(11 22 33))
(map (lambda (x) (* 2 x)) xs)
```


Running Scheme programs

- Some Scheme implementations:
  - PLT Scheme
  - Jaffer's SCM
  - MIT/GNU Scheme
  - Chez Scheme (commercial license)
  - Petite Chez Scheme
  - More at http://schemers.org/ > implementation
- Documentation, lots, among which:
  - Revised^6 Report on the Algorithmic Language Scheme, at http://www.r6rs.org/
  - The Scheme Programming Language, at http://www.scheme.com/tspl3/

Closures in C# (since 2.0)

- C# 2.0 anonymous function syntax

```
delegate int I2I(int x);
```

type I2I = int->int

```
I2I f = delegate(int x) { return x*3+7; };
int res = f(32);
```

Closure

- C# 3.0 lambda syntax

```
Func<int,int> f = x => x*3+7;
int res = f(32);
```

Type int->int

Closure

Closures may capture variables

- Local variable y is captured in delegate:

```
I2I MakeAdd(int y) {
  return delegate(int x) { return x+y; };
}
```

```
I2I g = MakeAdd(7);
int res = g(42);
```

C#: Captured variables may be updated

- Apply **f** to every element of the array:

```
void Apply(int[] arr, I2I f) {
  foreach (int x in arr)
    f(x);
}
```

- Compute the sum of array's elements:

```
void M() {
  int[] arr = ...
  int sum = 0;
  Apply(arr, delegate(int x) { sum += x; });
  Console.WriteLine(sum);
}
```

C#: Captures lvalue of variable **sum**

Closures in Java (since 1.2)

- Use anonymous inner classes

```
interface I2I {
  int invoke(int x);
}
```

type I2I = int->int

```
I2I f =
  new I2I() {
    public int invoke(int x) {
      return x*3+7;
    }
  };
int res = f.invoke(32);
```

Closure

More precisely: Instance of anonymous inner class that implements I2I

A Java closure capturing variables

- Captured variables must be final:

```
I2I MakeAdd(final int y) {
  return new I2I() {
    public int invoke(int x) { return x+y; }
  };
}
```

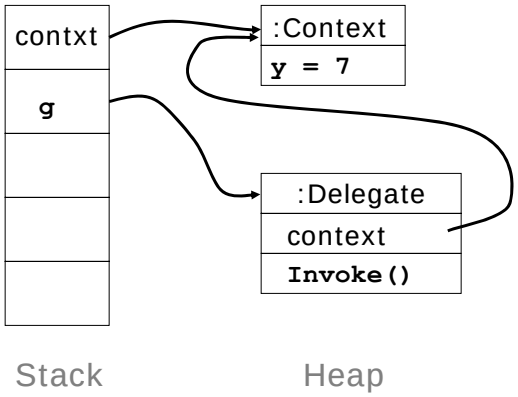
```
I2I g = MakeAdd(7);
int res = g.invoke(42);
```

Java: Only the rvalue of y is captured

- So using an `Apply(int[] arr, I2I f)` method in Java is more cumbersome – exercise

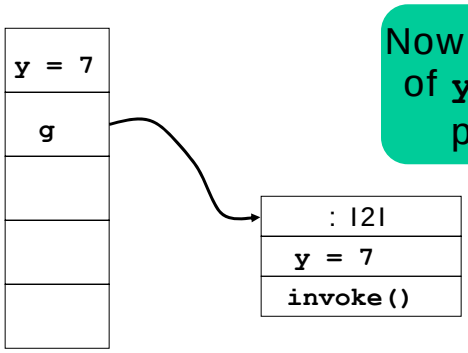
Implementation of C# closures

- A C# closure captures the lvalue:
  - stores the variable in a context object on heap
  - both method and closure use the context object



Implementation of Java closures

- A Java closure captures the rvalue:
  - stores the variable's value in the closure
  - method and closure each have a copy

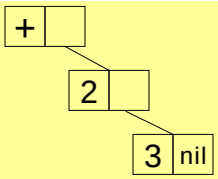


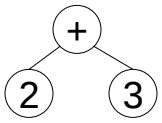
Now two copies of **y**; is that a problem?

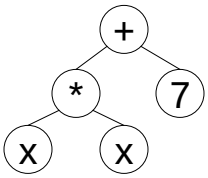
Back to Scheme: Abstract syntax and `eval`

- Abstract syntax for `(+ 2 3)` is `'(+ 2 3)`
- Abstract syntax can be evaluated by `eval`

```
> (+ 2 3)
5
> '(+ 2 3)
(+ 2 3)
> (eval '(+ 2 3))
5
> (define myexpr '(+ (* x x) 7))
> myexpr
(+ (* x x) 7)
> (define x 10)
> (eval myexpr)
107
```







Constructing abstract syntax

- Abstract syntax can be built with list and quote:

```
> (define e '(* x 3))
> (list '+ e '7)
(+ (* x 3) 7)
> (define (addsqr y)(list '+ 'x (* y y)))
> (addqrgen 7)
(+ x 49)
> (define (addsqrdef y)
      (list 'define '(f x)
        (list '+ 'x (* y y))))
> (addsqrdef 7)
(define (f x) (+ x 49))
```

Build AST for function that adds y^2 to x

Scheme quasiquotation: Comma and backquote

- Using `list` and quote can be confusing
- Backquote and comma make life easier
  - Backquote quotes everything except subexpressions preceded by comma
  - Comma "unquotes" a subexpression

```
> (define (addsqrdef y)
      `(define (f x) (+ x ,(* y y))))
> (addsqrdef 7)
(define (f x) (+ x 49))
```

Generator of specialized power power(n,x) functions

```
(define (powergen n)
  (if (> n 0)
      (if (eq? (remainder n 2) 0)
          `(sqr ,(powergen (/ n 2)))
          `(* x ,(powergen (- n 1)))
          )
      `1)
  )

(define (mkpower n)
  (eval `(define (pow x) ,(powergen n))))
```


Two-level languages and binding-times

- Scheme with backquote and comma is a two-level language:
  - Backquote: dynamic (late) computation
  - Comma: static (early) computation in dynamic context

```
(define (power n x)
  (if (> n 0)
      (if (eq? (remainder n 2) 0)
          (sqr (power (/ n 2) x))
          (* x (power (- n 1) x))
          )
      1)
  )
```

Ten-minute exercise

- Ex 1: Use backquote and comma to write an expression that builds

  `(+ y 2^97)`

  where the value of 2^{97} must be computed and inserted

- Ex 2: Assume x is static and y dynamic in

  `(+ (* 11 x) (* y 22))`

  – Mark static and dynamic parts (green, red)
  – Write expression that builds the above for any given value of x

Partial evaluation

- Proposed by Yoshihiko Futamura 1970
- Initially studied in USSR and Sweden
- Idea:
  – Assume p is a two-input program p(in1,in2)
  – But we have only part of the input, in1
  – Then we cannot run (evaluate) program p
  – But can partially evaluate it [mix](p,in1)
  – We don't get a result, but a new program r
  – Running r on in2 will then give the result
- Two-stage execution ...

Interpreter, compiler and partial evaluator (mix)

- Let p be a program and inp input data
- Running p on input inp gives output out
  - In symbols: [p](inp) = out
- An interpreter int is a program such that
  - [int](p, inp) = out
- A compiler comp is a program such that
  - If target = [comp](p) then [target](inp) = out
- Now let p be a two-input program, [p](in1,in2) = out
- A partial evaluator is a program mix such that
  - If r = [mix](p,in1) then [r](in2) = out
- The partial evaluator runs p on only part of its input, giving a residual program r

The three Futamura projections

- First: compilation
  - If target = [mix](int,p)
    then [target](d) = r
- Second: compiler generation
  - If comp = [mix](mix,int)
    then [comp](p) = t
- Third: compiler generator generation
  - If cogen = [mix](mix,mix)
    then [cogen](int) = comp
- There's no Fourth Futamura projection:
  - Because [cogen](mix) = cogen

This works in practice

- Self-application is non-trivial in practice
- First success 1985 by Jones, Sondergaard, Sestoft at DIKU
- Had to invent binding-time analysis
- See Jones, Gomard, Sestoft: *Partial Evaluation and Automatic Program Generation*, Prentice-Hall Intl. 1993
- Try the Scheme0 partial evaluator

Runtime code generation in C#

- Could generate C# code, then call compiler
- But compiler must be installed, big overhead
- Better generate .NET bytecode directly
- Example: To generate

```
public static int MyMethod(int x) {
  Console.WriteLine("MyMethod() was called");
  return x + 42;
}
```

- Use an ILGenerator **`ilg`** to make the body

```
ilg.EmitWriteLine("MyMethod() was called");
ilg.Emit(OpCodes.Ldarg_0);
ilg.Emit(OpCodes.Ldc_I4, 42);
ilg.Emit(OpCodes.Add);
ilg.Emit(OpCodes.Ret);
```

The full story

- The ILGenerator belongs to a DynamicMethod

```
DynamicMethod methodBuilder =
  new DynamicMethod("MyMethod",
                    typeof(int),
                    new Type[] { typeof(int) },
                    typeof(String).Module);
ILGenerator ilg = methodBuilder.GetILGenerator();
... use ilg as on previous slide ...
```

- Creating and calling the new method:

```
int res;
I2I mm = (I2I)methodBuilder.CreateDelegate(typeof(I2I));
res = mm(17);
```

`public delegate int I2I(int x);` as before