Programs as data

Scheme and program generation

Peter Sestoft

2012-11-26

Today

- Program generation
  - Programs that generate programs
- The Scheme programming language
  - Dynamically typed functional language
  - Concrete syntax = abstract syntax
- Program generation in Scheme
  - Two-level languages
  - Distinguishing binding-times
- Partial evaluation: automatic program specialization

The power(n, x) function

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

```
static double Power(int n, double x) {
  double 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;
}
```

Example:

3^5
$= 3 * 3^4$
$= 3 * (3^2)^2$
$= 3 * 9^2$
$= 3 * 81$
$= 243$

Specialized power(n,x) for n=5

- Assume we need to compute x^5 for many x
- So we know, statically, that n=5, but don't know x
- Then a *specialized* function Power_5(x)
  - would compute exactly the same result
  - but would be faster (why?)

```
static double Power_5(double x) {
  double 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("static double Power_" + n + "(double x) {");
  System.out.println("  double 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

```
static double Power(int n, double x) {
  double 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 generates the red code
- But tiresome to generate code as text

The Scheme language

- Design by Guy L Steele 1978
  - Plus a revolutionary compilation technique
  - Master's thesis from MIT
  - Co-author of *Java Language Specification*
  - Designer of other languages: Common Lisp Object System (CLOS), Fortress, ...
- Scheme descends from Lisp (McCarthy 1960)
- A higher-order functional language
- Like Scala, ML, F# but no static types
- Very simple syntax, lots of parentheses

Scheme expressions

- Compute 7+9:

```
(+ 7 9)
```

- Compute 7*9+13:

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

- 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))
```

Prefix notation

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

```
fun x -> x*3+7
```

F#, Ocaml

```
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, Scala

```
λx . x*3+7
```

Lambda calculus, 1936

Closures in Scheme

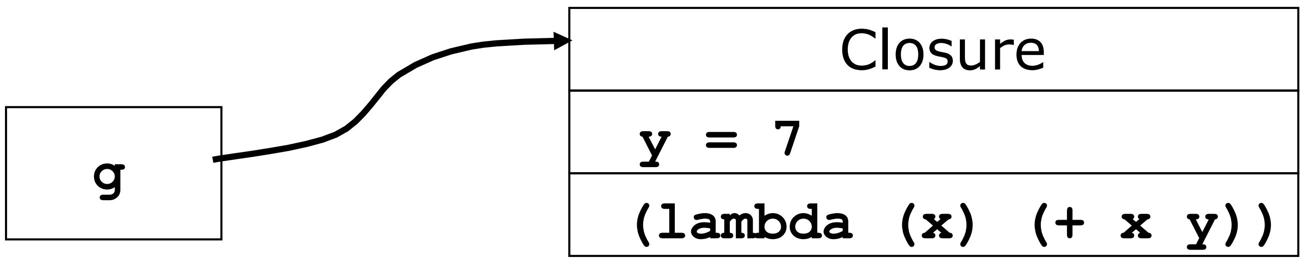
- An anonymous function may use a variable from an enclosing scope

```
(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:

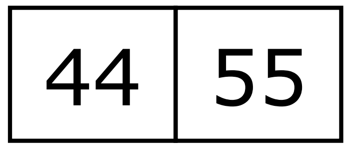
```
(car (cdr xs))
```

Pairs and lists: s-expressions

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

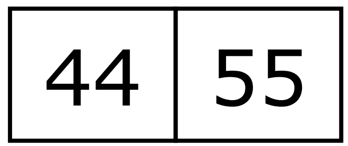
- Creating a new cons cell:

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



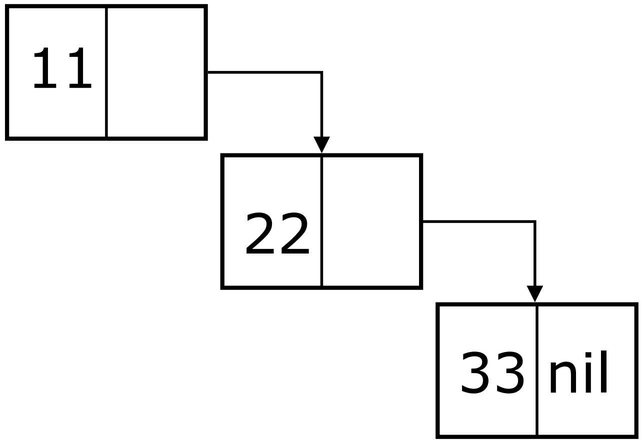
- A constant cons cell:

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



A list is a special s-expression

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



- Four ways to build that 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 only the third element
  - for computing the sum of the first and second element
  - for testing whether first element is positive
- Write a Scheme expression corresponding to 11+x*(22+x*(33+x*0))

Some list-processing functions

- Length of a list:

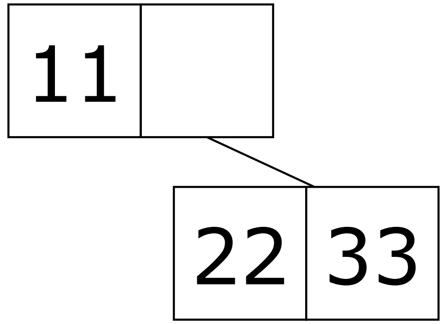
```
(define (len xs)
  (if (null? xs)
      0
      (+ 1 (len (cdr xs)))
))
```

- 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)
```

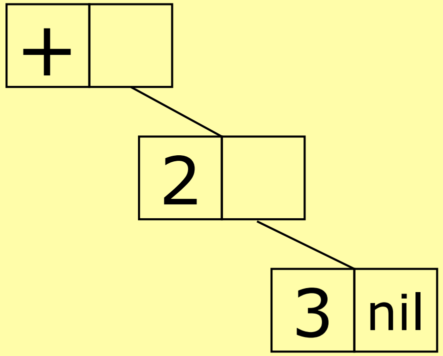
Running Scheme programs

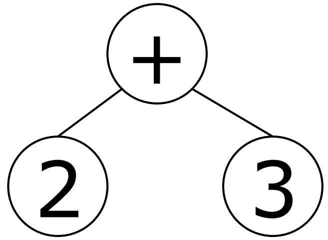
- Some Scheme implementations:
  - Gambit Scheme, http://dynamo.iro.umontreal.ca/~gambit
  - MIT/GNU Scheme (Linux, MacOS, Win)
  - Jaffer's SCM (only Linux and other Unixes)
  - Racket (formerly PLT Scheme)
  - Chez Scheme (commercial)
  - More at http://schemers.org/ > implementation
- Documentation, 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/

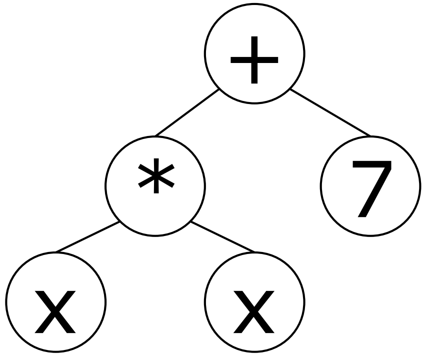
Data representing expressions: Abstract syntax and the `eval` function

- Abstract syntax for **`(+ 2 3)`** is **`'(+ 2 3)`**
- Abstract syntax can be evaluated by built-in **`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 ee '(* x 3))
> (list '+ ee '7)
(+ (* x 3) 7)

> (define (addsqr y) (list '+ 'x (* y y)))
> (addsqr 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

AST only

From AST to defined function

- The abstract syntax tree (AST) is just data
- To define a function, we must **`eval`** the AST
  - Just like **`javac`** followed by **`java`**; here just one step

```
> (addsqrdef 7)
(define (f x) (+ x 49))

> (f 10)
# ... undefined identifier: f

> (eval (addsqrdef 7))

> (f 10)
59
```

Build AST only

Cannot call f

Build and eval

Can call f

Scheme quasiquotation: Comma and backquote

- Using `list` and quote can be confusing
- Backquote and comma make life easier
  - Backquote “quotes” everything so it gets constructed, not evaluated
  - Comma “unquotes” a subexpression so it gets evaluated, not constructed

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

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 (using **`power`**) and inserted as number

- 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
- Studied in USSR and Sweden in the 1970'es
- 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**, or **specialize**, it
    r = [spec](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 (spec)

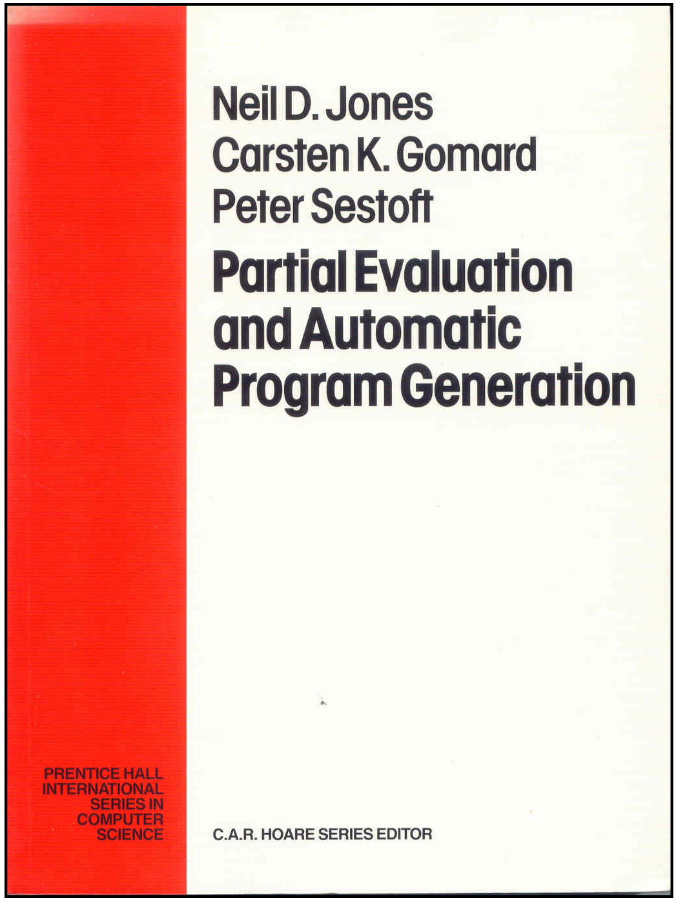
- Let s be a source program and inp input data
- Running s on input inp gives output out
  - In symbols: [s](inp) = out
- An *interpreter* int is a program such that
  - [int](s, inp) = out
- A *compiler* comp is a program such that
  - **If** target = [comp](s) **then** [target](inp) = out
- Now let p be a two-input program, [p](in1,in2) = out
- A *partial evaluator* is a program spec such that
  - **If** r = [spec](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 = [spec](int,s)
    then [target](in) = out
- Second: compiler generation
  - If comp = [spec](spec,int)
    then [comp](s) = target
- Third: compiler generator generation
  - If cogen = [spec](spec,spec)
    then [cogen](int) = comp
- There’s no Fourth Futamura projection:
  - Because [cogen](spec) = cogen

This actually works!

- Self-application is non-trivial in practice
  - Avoid generating large trivial specializations
- Success 1985 at University of Copenhagen
- We invented *binding-time analysis*, a static analysis:
  - Which computations depend only on known data
  - Those may be performed at specialization time



http://www.itu.dk/people/sestoft/pebook/pebook.html

Example function: (power n x)

```
((define (pow n x)
   (if (op equal? n 0)
     1
     (if (op even? n)
       (call pow (op quotient n 2) (op * x x))
       (op * x (call pow (op - n 1) x))))))
```

"Scheme0" version of (power n x)

```
(define pa2 (monotate power '(s d)))
```

```
((define (((pow 2) s d) (n) (x))
  (ifs (ops equal? n 0)
     (lift 1)
     (ifs (ops even? n)
          (calld ((pow 2) s d)
                 ((ops quotient n 2))
                 ((opd * x x)))
          (opd * x (calls ((pow 2) s d) ((ops - n 1)) (x)))))))
```

After binding-time analysis and annotation (s=static, d=dynamic)

Specializing for n=97

```
> (scheme (spec pa2 '(97)))
```

Specialize wrt n=97

```
((define (pow*sd-1 x) (* x (pow*sd-2 (* x x))))
 (define (pow*sd-2 x) (pow*sd-3 (* x x)))
 (define (pow*sd-3 x) (pow*sd-4 (* x x)))
 (define (pow*sd-4 x) (pow*sd-5 (* x x)))
 (define (pow*sd-5 x) (pow*sd-6 (* x x)))
 (define (pow*sd-6 x) (* x (pow*sd-7 (* x x))))
 (define (pow*sd-7 x) (* x '1)))
```

Result

Compile and run it

```
> (make 'power97 (spec pa2 '(97)))
> (power97 3)
19088056323407827075424486287615602692670648963
```

Generating and using a specializer

```
> (define sann (monotate specializer '(s d)))
> (define sp2 (spec sann (list pa2)))
```

Specialize specializer wrt power

Resulting sp2 power specializer

```
((define (specialize*sd-1 vs0)
   (complete*dds-1
     (cons (cons '((pow 2) s d) vs0) '())
     '()))
 (define (complete*dds-1 pending marked)
   (if (null? pending)
     '()
     (generate*dsdds-1 (car pending) pending marked)))
 (define (generate*dsdds-1 fvs pending marked)
   (if (equal? '((pow 2) s d) (car fvs))
     (gen1*sdddds-1
       (reduce*ssdsds-1 (cdr fvs) '(x))
       fvs
       pending
       (cons fvs marked))
     (generate*dsdds-2 fvs pending marked)))
 (define (reduce*ssdsds-1 vs vd)
   (if (evalbase 'equal? (cons (car vs) (cons '0 '())))
     (reduce*ssdsds-2 vs vd)
     (reduce*ssdsds-3 vs vd)))
 (define (reduce*ssdsds-2 vs vd)
   (cons 'quote (cons '1 '())))
 (define (reduce*ssdsds-3 vs vd)
   (if (evalbase 'even? (cons (car vs) '()))
     (reduce*ssdsds-4 vs vd)
     (reduce*ssdsds-5 vs vd)))
 (define (reduce*ssdsds-4 vs vd)
   (cons 'call
         (cons (cons '((pow 2) s d)
                     (cons (evalbase
                             'quotient
                             (cons (car vs) (cons '2 '())))
                           '()))
               (cons (cons 'op
                           (cons '* (cons (car vd) (cons (car vd) '()))))
                     '()))))
 (define (reduce*ssdsds-5 vs vd)
   (cons 'op
         (cons '*
               (cons (car vd)
                     (cons (docalls*ssdd-1
                             (cons (evalbase '- (cons (car vs) (cons '1 '())))
                                   '())
                             (cons (car vd) '()))
                           '())))))
 (define (docalls*ssdd-1 args argd)
   (if (nodup '(2) argd)
     (reduce*ssdsds-1 args argd)
     (cons 'call
           (cons (cons '((pow 2) s d) args) argd))))
 (define (gen1*sdddds-1 evs fvs pending newmarked)
   (cons (cons 'define
               (cons (cons fvs '(x)) (cons evs '())))
         (complete*dds-1
           (diff (successors evs pending) newmarked)
          newmarked)))
 (define (generate*dsdds-2 fvs pending marked)
   (error 'generate
          '"Undefined function: ~s"
          (car fvs))))
```

Compile and use it

```
> (make 'powergen sp2)
> (powergen '(97))
```

Result is as before

```
((define (pow*sd-1 x) (* x (pow*sd-2 (* x x))))
 (define (pow*sd-2 x) (pow*sd-3 (* x x)))
 (define (pow*sd-3 x) (pow*sd-4 (* x x)))
 (define (pow*sd-4 x) (pow*sd-5 (* x x)))
 (define (pow*sd-5 x) (pow*sd-6 (* x x)))
 (define (pow*sd-6 x) (* x (pow*sd-7 (* x x))))
 (define (pow*sd-7 x) (* x '1)))
```

Generating a specializer generator

```
> (define cc (spec sann (list sann)))
```

Specialize specializer wrt itself

Use resulting specializer generator to generate a power specializer

```
> (make 'cogen cc)
> (define cp2 (cogen (list pa2)))
```

Result cp2 is identical to sp2

Regenerate specializer generator

```
> (define ccc (cogen (list sann)))
> (equal? cc ccc)
#t
```

Result ccc is identical to cc

Pitfalls of partial evaluation

- Sometimes nothing can be specialized
  - Eg x static (known) but n dynamic in power(n,x)
  - Static computation under dynamic control, dangerous
- Infinitely large "specialized" programs
- Very large specialized programs, no speedup
  - Subsequent compilation/code generation slow
  - Many instruction cache misses
- Inadequate binding-time separation
  - Variables that "should" be static become dynamic
- Lots of research on these problems since 1985
  - For Scheme, Prolog, C, ML, Java, C#, ...
  - The toy Scheme0 specializer used here is very simple
- Specialization, spreadsheets, graphics processors