Advanced Models and Programs

Scheme and program generation

Peter Sestoft
Monday 2010-02-22

Today

• Program generation
  – Programs that generate programs
• The Scheme programming language
  – Dynamically typed functional language
  – Concrete syntax = abstract syntax
• Informal course evaluation
• Program generation in Scheme
  – Two-level languages
  – Distinguishing binding-times
• Partial evaluation = program specialization
The power($n$, $x$) function

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

```java
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:

$3^5$

$= 3 \times 3^4$

$= 3 \times (3^2)^2$

$= 3 \times 9^2$

$= 3 \times 81$

$= 243$

Specialized power($n$,x) for n=5

• What if we must 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?)

```java
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

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

Binding times in Power(n,x)

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

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

- The generator performs the green code and prints (emits) 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*
  - Now designing a new language “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 \(x-15\):
  \[
  (if (< x 15) (* x x) (- x 15))
  \]
Scheme function definitions

- Defining the function f(x) = x*3+7:
  
  \[
  \text{(define } (f \ x) \ (+ \ (* \ x \ 3) \ 7))
  \]

- Same in C/C++/Java/C#:
  
  \[
  \text{int } f(\text{int } x) \{ \text{ return } x*3+7; } \}
  \]

- Calling the function on argument 10:
  
  \[
  (f 10)
  \]

- Defining a recursive function:
  
  \[
  \text{(define } (fac } n) \)
  
  \[
  \quad \text{(if } (= \ n \ 0)
  
  \quad \quad 1
  
  \quad \quad (* \ n \ (fac (- \ n \ 1))))
  \]

The power function in Scheme

\[
\text{(define } (sqr } x) \ (* \ x \ x))
\]

\[
\text{(define } (power } n \ x)
\]

\[
\quad \text{(if } (> \ n \ 0)
\]

\[
\quad \quad \text{(if } (\text{eq? } (\text{remainder } n \ 2) \ 0)
\]

\[
\quad \quad \quad (\text{sqr } (\text{power } (/ \ n \ 2) \ x))
\]

\[
\quad \quad \quad (* \ x \ (\text{power } (- \ n \ 1) \ x))
\]

\[
\quad \quad 1)
\]

\[
\quad)
\]

\[
> \ (\text{power } 10 \ 2)
\]

\[
1024
\]

\[
> \ (\text{power } 97 \ 2)
\]

\[
158456325028528675187087900672
\]
Scheme anonymous functions

- The anonymous function \( x \rightarrow x \times 3 + 7 \)
  
  \[
  (\text{lambda} \ (x) \ (+ \ (* \ x \ 3) \ 7))
  \]

- Applying the function to argument 10:
  
  \[
  ((\text{lambda} \ (x) \ (+ \ (* \ x \ 3) \ 7)) \ 10)
  \]

- Anonymous functions in other languages
  
  - \( \text{fun} \ x \rightarrow x \times 3 + 7 \)  
    - F#, Ocaml
  
  - \( \text{fn} \ x \Rightarrow x \times 3 + 7 \)  
    - Standard ML, 1978
  
  - \text{delegate}(\text{int} \ x) \ { \text{return} \ x \times 3 + 7; \ }
    - C# 2.0
  
  - \( x \Rightarrow x \times 3 + 7 \)
    - C# 3.0, Scala
  
  - \( \lambda x . x \times 3 + 7 \)
    - Lambda calculus, 1936

Closures in Scheme

- An anonymous function may use a variable from an enclosing scope
  
  \[
  (\text{define} \ (\text{makeadd} \ y)\n   \ (\text{lambda} \ (x) \ (+ \ x \ y))\n  )
  \]

- A closure must be built for the function:
  
  \[
  (\text{define} \ g \ (\text{makeadd} \ 7))
  \]

  \[
  (g \ 42)
  \]

  g’s value is a closure

<table>
<thead>
<tr>
<th>Closure</th>
</tr>
</thead>
<tbody>
<tr>
<td>y = 7</td>
</tr>
<tr>
<td>(lambda (x) (+ x y))</td>
</tr>
</tbody>
</table>
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:
  \[\texttt{'(11 22 33)}\]
- Defining \texttt{xs} to be that list:
  \[\texttt{(define xs '(11 22 33))}\]
- The first element of \texttt{xs}:
  \[\texttt{(car xs)}\]
- The rest of \texttt{xs}:
  \[\texttt{(cdr xs)}\]
- The second element of \texttt{xs}:
  \[\texttt{(car (cdr xs))}\]

Pairs and lists: s-expressions

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

  ![Diagram of cons cell with car and cdr](image)

- Creating a new cons cell:
  \[\texttt{(cons 44 (+ 44 11))}\]
  \[\texttt{44 55}\]
- A constant cons cell:
  \[\texttt{'(44 . 55)}\]
  \[\texttt{44 55}\]
A list is a special s-expression

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

```
  11
  
22
  
33 nil
```

- 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:

  \[
  \text{(define (len xs)}
  \begin{align*}
  &\text{if (null? xs)} \\
  &\quad 0 \\
  &\quad (+ 1 \text{ (len (cdr xs))})
  \end{align*}
  \text{)}
  \]

- Sum of a list’s elements:

  \[
  \text{(define (sum xs)}
  \begin{align*}
  &\text{if (null? xs)} \\
  &\quad 0 \\
  &\quad (+ \text{ (car xs)} \text{ (sum (cdr xs))})
  \end{align*}
  \text{)}
  \]

Some tree-processing functions

- Representing a tree:

  \[
  \text{(define t}
  \begin{align*}
  &\text{'(11 . (22 . 33))}
  \end{align*}
  \text{)}
  \]

- Depth of a tree:

  \[
  \text{(define (depth t)}
  \begin{align*}
  &\text{if (pair? t)} \\
  &\quad (+ 1 \text{ (max (depth (car t))}) \\
  &\quad \quad \text{ (depth (cdr t))}) \\
  &\quad 0
  \end{align*}
  \text{)}
  \]
Higher-order functions

• Mapping a function over a list:

\[
\begin{align*}
\text{(define } & (\text{map } f \ \text{xs}) \\
& \text{(if } (\text{null? } \text{xs}) \\
& \quad () \\
& \quad (\text{cons } (f \ (\text{car } \text{xs})) \ (\text{map } f \ (\text{cdr } \text{xs}))))
\end{align*}
\]

• Example use:

\[
\begin{align*}
\text{(define } \text{xs }' (11 \ 22 \ 33)) \\
\text{(map } & (\text{lambda } (x) \ (* \ 2 \ x)) \ \text{xs})
\end{align*}
\]

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/
Data representing expressions: Abstract syntax and the eval function

- Abstract syntax for \((+ 2 3)\) is \(\texttt{(+ 2 3)}\)
- Abstract syntax can be evaluated by \texttt{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)))
> (addsqr 7) (+ x 49)
> (define (addsqrdef y)
> (list 'define 'f x)
> (list '+ 'x (* y y)))
> (addsqrdef 7)
> (define (f x) (+ x 49))
```
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

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

Generator of specialized power power(n,x) functions

```scheme
(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
  
  (+ (* 11 x) (* y 22))
  
  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
- 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
    \( r = \text{[spec]}(p, in1) \)
  - 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 \( p \) 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 \( r = [\text{spec}](p, in1) \) 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,p)
    then [target](d) = r

- Second: compiler generation
  - If comp = [spec](spec,int)
    then [comp](p) = t

- 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

- 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: \((\text{power } n \ x)\)

\[
\begin{align*}
((\text{define} \ (\text{pow } n \ x)) & \quad (\text{if} \ (\text{op} \ \text{equal?} \ n \ 0) \\
& \quad 1 \\
& \quad (\text{if} \ (\text{op} \ \text{even?} \ n) \\
& \quad \quad (\text{call} \ \text{pow} \ (\text{op} \ \text{quotient} \ n \ 2) \ (\text{op} \ \ast \ x \ x)) \\
& \quad \quad (\text{op} \ \ast \ x \ (\text{call} \ \text{pow} \ (\text{op} \ - \ n \ 1) \ x)))))) \\
\end{align*}
\]

\((\text{define} \ (\text{((pow } 2) \ s \ d) \ (n) \ (x))) \quad (\text{ifs} \ (\text{ops} \ \text{equal?} \ n \ 0) \quad \text{lift} \ 1) \\
\quad (\text{ifs} \ (\text{ops} \ \text{even?} \ n) \\
\quad \quad (\text{call} \ ((\text{pow } 2) \ s \ d) \ (\text{ops} \ \text{quotient} \ n \ 2) \ (\text{ops} \ \ast \ x \ x)) \\
\quad \quad (\text{op} \ \ast \ x \ (\text{calls} \ ((\text{pow } 2) \ s \ d) \ ((\text{ops} \ - \ n \ 1) \ (x))))))))

After binding-time analysis and annotation

Specializing for \(n=97\)

\[
\begin{align*}
> \quad \text{(pretty} \ \text{(scheme} \ \text{spec pa2 } '(97)))\text{)}) \\
\end{align*}
\]

\[
\begin{align*}
((\text{define} \ (\text{pow}^*sd-1 \ x) & \quad (\ast \ x \ (\text{pow}^*sd-2 \ (\ast \ x \ x)))) \\
(\text{define} \ (\text{pow}^*sd-2 \ x) & \quad (\text{pow}^*sd-3 \ (\ast \ x \ x))) \\
(\text{define} \ (\text{pow}^*sd-3 \ x) & \quad (\text{pow}^*sd-4 \ (\ast \ x \ x))) \\
(\text{define} \ (\text{pow}^*sd-4 \ x) & \quad (\text{pow}^*sd-5 \ (\ast \ x \ x))) \\
(\text{define} \ (\text{pow}^*sd-5 \ x) & \quad (\text{pow}^*sd-6 \ (\ast \ x \ x))) \\
(\text{define} \ (\text{pow}^*sd-6 \ x) & \quad (\ast \ x \ (\text{pow}^*sd-7 \ (\ast \ x \ x)))) \\
(\text{define} \ (\text{pow}^*sd-7 \ x) & \quad (\ast \ 1)))
\end{align*}
\]

\[
\begin{align*}
> \quad \text{(make} \ \text{'power97} \ \text{(spec pa2 } '(97)))\text{)} \\
> \quad \text{(power97} \ 3) \\
19088056323407827075424486287615602692670648963
\end{align*}
\]
Generating and using a specializer

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

Compile and use it

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

Resulting sp2 power specializer

Specialize specializer wrt power

Generating a specializer generator

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

Use resulting specializer generator to generate a power specializer

> (make 'cogen cc)
> (define cp2 (cogen (list pa2)))
> (equal? cp2 sp2)

Result cp2 is identical to sp2

Regenerate specializer generator

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

Result ccc is identical to cc
What's next

- Friday 26 Feb: Runtime code generation in (Java and) C#
- Monday 1 Mar: Student presentations of Python, Erlang, F#, Haskell
- Friday 5 Mar onwards: Extended static checking and dataflow analysis, by Joe Kiniry

- Exercises week 5: Scheme, program generation in Scheme, and runtime code generation in C#/CLI