CSE341: Programming Languages

Structs, Implementing Languages, Implementing Higher-Order Functions

Alan Borning
(slides “borrowed” from Dan Grossman)
Fall 2012

An aside - Parenthesis bias

• If you look at the HTML for a web page, it takes the same approach:
  • <foo> written <foo>
  • ) written </foo>

• But for some reason, LISP/Scheme/Racket is the target of subjective parenthesis-bashing
  • Curiously, often by people who have no problem with HTML
  • You are entitled to your opinion about syntax, but a good historian wouldn’t refuse to study a country where he/she didn’t like people’s accents

Fall 2012

Review

• Given pairs and dynamic typing, you can code up “one-of types” by using first list-element like a constructor name:

  ```racket
  (define (const i)   (list 'const i))
  (define (add e1 e2) (list 'add e1 e2))
  (define (negate e)  (list 'negate e))
  ```

• But much better and more convenient is Racket’s structs
  • Makes a new dynamic type (pair? answers false)
  • Provides constructor, predicate, accessors

```racket
(struct const (i) #:transparent)
(struct add (e1 e2) #:transparent)
(struct negate (e) #:transparent)
```

Haskell’s view of Racket’s “type system”

One way to describe Racket is that it has “one big datatype”
  • All values have this same one type
  • Constructors are applied implicitly (values are tagged)
  • 42 is implicitly “int constructor with 42”
  • Primitives implicitly check tags and extract data, raising errors for wrong constructors
  • * is implicitly “check for int constructors and extract data”
  • [Actually Racket has a numeric tower that * works on]
  • Built-in: numbers, strings, booleans, pairs, symbols, procedures, etc.
  • Each struct creates a new constructor, a feature many dynamic languages do not have
  • (struct ...) can be neither a function nor a macro

Define trees

• Either lists or structs (we’ll use structs) can then let us build trees to represent compound data such as expressions

  ```racket
  (add (const 4)
       (negate (add (const 1)
                    (negate (const 7))))))
  ```

• Since Racket is dynamically typed, the idea that a set of constructors are variants for “an expression datatype” is in our heads / comments

http://xkcd.com/297/

  • Invented garbage collection

Haskell’s view of Racket’s “type system”

One way to describe Racket is that it has “one big datatype”
  • All values have this same one type
  • Constructors are applied implicitly (values are tagged)
  • 42 is implicitly “int constructor with 42”
  • Primitives implicitly check tags and extract data, raising errors for wrong constructors
  • * is implicitly “check for int constructors and extract data”
  • [Actually Racket has a numeric tower that * works on]
  • Built-in: numbers, strings, booleans, pairs, symbols, procedures, etc.
  • Each struct creates a new constructor, a feature many dynamic languages do not have
  • (struct ...) can be neither a function nor a macro

Define trees

• Either lists or structs (we’ll use structs) can then let us build trees to represent compound data such as expressions

  ```racket
  (add (const 4)
       (negate (add (const 1)
                    (negate (const 7))))))
  ```

• Since Racket is dynamically typed, the idea that a set of constructors are variants for “an expression datatype” is in our heads / comments

http://xkcd.com/297/

  • Invented garbage collection
Implementing PLs

Most of the course is learning fundamental concepts for using PLs
- Syntax vs. semantics vs. idioms
- Powerful constructs like pattern-matching, closures, dynamically typed pairs, macros, ...

An educated computer scientist should also know some things about implementing PLs
- Implementing something requires fully understanding its semantics
- Things like closures and objects are not “magic”
- Many programming tasks are like implementing PLs
  - Example: rendering a document (“program” is the structured document and “pixels” is the output)

Ways to implement a language

Two fundamental ways to implement a PL A
- Write an interpreter in another language B
  - Better names: evaluator, executor
  - Take a program in A and produce an answer (in A)
- Write a compiler in another language B to a third language C
  - Better name: translator
  - Translation must preserve meaning (equivalence)

We call B the metalanguage; crucial to keep A and B straight

Very first language needed a hardware implementation

Reality more complicated

Evaluation (interpreter) and translation (compiler) are your options
- But in modern practice have both and multiple layers

A plausible example:
- Java compiler to bytecode intermediate language
- Have an interpreter for bytecode (itself in binary), but compile frequent functions to binary at run-time
- The chip is itself an interpreter for binary
  - Well, except these days the x86 has a translator in hardware to more primitive micro-operations that it then executes

Racket uses a similar mix

Implementing a language

"(fn x => x + x) 7"

Possible Errors / warnings

Parsing

Call

Function

Negate

Constant

Var

Var

4

Static checking

(what checked depends on PL.)

Rest of implementation

Possible Errors / warnings

Skipping those steps

Alternately, we can embed our language inside (data structures) in the metalanguage
- Skip parsing: Use constructors instead of just strings
- These abstract syntax trees (ASTs) are already ideal structures for passing to an interpreter

We can also, for simplicity, skip static checking
- Assume subexpressions are actually subexpressions
  - Do not worry about (add #f "hi")
- For dynamic errors in the embedded language, interpreter can give an error message
  - Do not worry about (add (fun ...) (int 14))

The arith-exp example

This embedding approach is exactly what we did for the PL of arithmetic expressions:

```
(struct const (i) #:transparent)
(struct add (e1 e2) #:transparent)
(struct negate (e) #:transparent)

(add (const 4)
  (negate (add (const 1)
    (negate (const 7)))))
```

Note: So simple there are no dynamic type errors in the interpreter
The interpreter

An interpreter takes programs in the language and produces values (answers) in the language
- Typically via recursive helper functions with cases
- This example is so simple we don't need a helper and can assume all recursive results are constants

```
(define (eval-exp e)
  (cond
    [(const? e) e]
    [(add? e)
     (const (+ (const-i (eval-exp (add-e1 e)))
              (const-i (eval-exp (add-e2 e)))))]
    [(negate? e)
     (const (- (const-i (eval-exp (negate-e e)))))]
    [#t (error "eval-exp expected an expression")])
```

“Macros”

Another advantage of the embedding approach is we can use the metalanguage to define helper functions that create programs in our language
- They generate the (abstract) syntax
- Result can then be put in a larger program or evaluated
- This is a lot like a macro, using the metalanguage as our macro system

Example:
All this does is create a program that has four constant expressions:
```
(define (triple x) (add x (add x x)))
(define p (add (const 1) (triple (const 2))))
```

What’s missing

Two very interesting features missing from our arithmetic-expression language:
- Local variables
- Higher-order functions with lexical scope

How to support local variables:
- Interpreter helper function(s) need to take an environment
- As we have said since lecture 1, the environment maps variable names to values
  - A Racket association list works well enough
  - Evaluate a variable expression by looking up the name
  - A let-body is evaluated in a larger environment

Higher-order functions

The “magic”: How is the “right environment” around for lexical scope when functions may return other functions, store them in data structures, etc.?

Lack of magic: The interpreter uses a closure data structure (with two parts) to keep the environment it will need to use later

Evaluate a function expression:
- A function is not a value; a closure is a value
- Create a closure out of (a) the function and (b) the current environment

Evaluate a function call:
```
• Evaluate 1st subexpression to a closure with current environment
• Evaluate 2nd subexpression to a value with current environment
• Evaluate closure’s function’s body in the closure’s environment, extended to map the function’s argument-name to the argument-value
  • And for recursion, function’s name to the whole closure
```

This is the same semantics we learned a few weeks ago “coded up”

Given a closure, the code part is only ever evaluated using the environment part (extended), not the environment at the call-site

Function calls

- Time to build a closure is tiny: a struct with two fields
- Space to store closures might be large if environment is large
  - But environments are immutable, so natural and correct to have lots of sharing, e.g., of list tails (cf. earlier lectures)
- Alternative: Homework 5 challenge problem is to, when creating a closure, store a possibly-smaller environment holding only the variables that are free variables in the function body
  - Free variables: Variables that occur, not counting shadowed uses of the same variable name
  - A function body would never need anything else from the environment

Is that expensive?

```
define (eval-exp e)
  (cond
    [(const? e) e]
    [(add? e)
     (const (+ (const-i (eval-exp (add-e1 e)))
              (const-i (eval-exp (add-e2 e)))))]
    [(negate? e)
     (const (- (const-i (eval-exp (negate-e e)))))]
    [#t (error "eval-exp expected an expression")])
```
Free variables examples

(lambda () (+ x y z))
(lambda (x) (+ x y z))
(lambda (x) (if x y z))
(lambda (x) (let ([y 0]) (+ x y z)))
(lambda (x y z) (+ x y z))
(lambda (x) (+ y (let ([y z]) (+ y y))))

Free variables examples

(lambda () (+ x y z)) ; x y z
(lambda (x) (+ x y z)) ; y z
(lambda (x) (if x y z)) ; y z
(lambda (x) (let ([y 0]) (+ x y z))) ; x
(lambda (x y z) (+ x y z)) ; {}
(lambda (x) (+ y (let ([y z]) (+ y y)))) ; y z

Free variables examples – mini-exercises

(lambda () (+ j 3))
((lambda (j) (+ j k 3))
(lambda (j) (let ([k 0]) (+ j k 3)))

Compiling higher-order functions

• Key to the interpreter approach: Interpreter helper function takes an environment argument
  – Recursive calls can use a different environment
• Can also compile higher-order functions by having the translation produce ‘regular’ functions (like in C or assembly) that all take an extra explicit argument called “environment”
• And compiler replaces all uses of free variables with code that looks up the variable using the environment argument
  – Can make these fast operations with some tricks
• Running program still creates closures and every function call passes the closure’s environment to the closure’s code