Evaluation and universal machines
- What is the role of evaluation in defining a language?
- How can we use evaluation to design a language?

The Eval/Apply Cycle
- Eval and Apply execute a cycle that unwinds our abstractions
  - Reduces to simple applications of built-in procedure to primitive data structures
- Key:
  - Evaluator determines meaning of programs (and hence our language)
  - Evaluator is just another program!!

Examining the role of Eval
- From perspective of a language designer
- From perspective of a theoretician

Eval from perspective of language designer
- Applicative order
- Dynamic vs. lexical scoping
- Lazy evaluation
  - Full normal order
  - By specifying arguments
  - Just for pairs
  - Decoupling analysis from evaluation

Eval is expensive (and not very clever)

```
(define (foo n)
  (cond ((= 0 n) . . .)
        (else (set! x (bar n))
              (foo (bar n))))
```

```
(define (fact n)
  (if (= n 1) 1 (* n (fact - n 1))))
```

Eval is expensive (and not very clever)

```
(define (foo n)
  (cond ((= 0 n) . . .)
        (else (set! x (bar n))
              (foo (bar n))))
```

```
(define (fact n)
  (if (= n 1) 1 (* n (fact - n 1))))
```

- Sure be nice to avoid this cost.
**static analysis: work done before execution**

- straight interpreter
  - expression \[\rightarrow\] interpreter \[\rightarrow\] value

- advanced interpreter or compiler
  - expr \[\rightarrow\] static analysis \[\rightarrow\] execution \[\rightarrow\] value

**Reasons to do static analysis**

- Improve execution performance
  - avoid repeating work if expression contains loops
  - simplify execution engine

- Catch common mistakes early
  - garbled expression
  - operand of incorrect type
  - wrong number of operands to procedure

- Prove properties of program
  - will be fast enough, won't run out of memory, etc.
  - significant current research topic

**Strategy of the analyze evaluator**

- analyze expr \[\rightarrow\] environment
  - expr \[\rightarrow\] analyze \[\rightarrow\] EP \[\rightarrow\] execution \[\rightarrow\] value

- Execution procedure
  - a scheme procedure
  - Env \[\rightarrow\] anytype

- analyze: expression \[\rightarrow\] (Env \[\rightarrow\] anytype)
  - (define (a-eval exp env) ((analyze exp) env))

**Implementing assignment**

- (define (analyze exp)
  (cond
   ((number? exp)   (analyze-number exp))
   ((variable? exp) (analyze-variable exp))
   ((assignment? exp) (analyze-assignment exp))
   ... ))

**Implementation of analyze-assignment**

- (define (analyze-assignment exp)
  (let ((var (assignmt-variable exp))
        (vproc (analyze (assignmt-value exp))))
   (lambda (env)
     (set-variable! var (vproc env) env))))

**Implementing number analysis**

- analyze-number is easy
  - (define (analyze-number exp) (lambda (env) exp))

- (black: analysis phase) (blue: execution phase)
Summary

- Output of analyze is an execution procedure
  - Given an environment
  - Produces value of expression
- Within analyze
  - Execution phase code appears inside
    `(lambda (env) ...)`
  - All other code runs during analysis phase

Subexpressions

`(analyze 'if (= n 1) 1 (* n (...) ))`

- Analysis phase:
  - `(analyze '(= n 1))` ➞ `pproc`
  - `(analyze 1)` ➞ `cproc`
  - `(analyze '(* n (...)))` ➞ `aproc`
- Execution phase
  - `(pproc env)` ➞ `#t` or `#f` (depending on `n`)
  - If `#t`, `(cproc env)`
  - If `#f`, `(aproc env)`

Implementation of `analyze-if`

```
(define (analyze-if exp)
  (let ((pproc (analyze (if-predicate exp)))
         (cproc (analyze (if-consequent exp)))
         (aproc (analyze (if-alternative exp))))
    (lambda (env)
      (if (true? (pproc env))
          (cproc env)
          (aproc env))))
```

Visualization of `analyze-if`

```
(if (= n 1)
  1
  (* n (...)))
```

Analyzing definitions

- Assume the following procedures for definitions like
  `(define x (+ y 1))`

```
(define-variable exp) x
(define-value exp) (+ y 1)
(define-variable! name value env) add binding to env
```

- Implement `analyze-definition`
  - The only execution-phase work is `define-variable!`
  - The definition-value might be an arbitrary expression

Implementation of `analyze-definition`

```
(define (analyze-definition exp)
  (let ((var (definition-variable exp))
         (vproc (analyze (definition-value exp))))
    (lambda (env)
      (define-variable! var (vproc env) env))))
```
Summary

• Within analyze
  • recursively call analyze on subexpressions
  • create an execution procedure which stores the EPs for subexpressions as local state

Implementing lambda

• Body stored in double bubble is an execution procedure

  • old make-procedure
    list<symbol>, expression, Env → Procedure

  • new make-procedure
    list<symbol>, (Env->anytype), Env → Procedure

(define (analyze-lambda exp)
  (let ((vars (lambda-parameters exp))
        (bproc (analyze (lambda-body exp))))
    (lambda (env)
      (make-procedure vars bproc env))))

Implementing apply: analysis phase

(define (analyze-application exp)
  (let ((fproc (analyze (operator exp)))
        (aprocs (map analyze (operands exp))))
    (lambda (env)
      (execute-application
        (fproc env)
        (map (lambda (aproc) (aproc env))
             aprocs))))

Implementing apply: execution phase

(define (execute-application proc args)
  (cond
   ((primitive-procedure? proc)
    ...)
   ((compound-procedure? proc)
    ((procedure-body proc)
     (extend-environment (parameters proc)
                      args
                      (environment proc))))
   (else ...)))

Summary

• In the analyze evaluator,
  • double bubble stores execution procedure, not expression

The dream of a universal machine...
The dream of a universal machine

ACME Universal machine
If you can say it, I can do it™

It wasn’t always this obvious

• “If it should turn out that the basic logics of a machine designed for the numerical solution of differential equations coincide with the logics of a machine intended to make bills for a department store, I would regard this as the most amazing coincidence that I have ever encountered”

Howard Aiken, writing in 1956 (designer of the Mark I "Electronic Brain", developed jointly by IBM and Harvard starting in 1939)

What is Eval really?

• We do describe devices, in a language called Scheme
• We have a machine that takes those descriptions and then behaves exactly as they specify
• Eval takes any program as input and reconfigures itself to simulate that input program
• EVAL is a universal machine

Why a Universal Machine?

• If EVAL can simulate any machine, and if EVAL is itself a description of a machine, then EVAL can simulate itself
  • This was our example of meval
• In fact, EVAL can simulate an evaluator for any other language
  • Just need to specify syntax, rules of evaluation
• An evaluator for any language can simulate any other language
  • Hence there is a general notion of computability – idea that a process can be computed independent of what language we are using, and that anything computable in one language is computable in any other language

Turing’s insight

• Alan Mathison Turing
• 1912-1954

Turing’s insight

• Was fascinated by Godel’s incompleteness results in decidability (1933)
  • In any axiomatic mathematical system there are propositions that cannot be proved or disproved within the axioms of the system
  • In particular the consistency of the axioms cannot be proved.
• Led Turing to investigate Hilbert’s Entscheidungsproblem
  • Given a mathematical proposition could one find an algorithm which would decide if the proposition was true or false?
  • For many propositions it was easy to find such an algorithm.
  • The real difficulty arose in proving that for certain propositions no such algorithm existed.
  • In general – Is there some fixed definite process which, in principle, can answer any mathematical question?
  • E.g., Suppose want to prove some theorem in geometry
    – Consider all proofs from axioms in 1 step
    – … in 2 steps ….
Turing’s insight

- Turing proposed a theoretical model of a simple kind of machine (now called a Turing machine) and argued that any “effective process” can be carried out by such a machine
- Each machine can be characterized by its program
- Programs can be coded and used as input to a machine
- Showed how to code a universal machine
- Wrote the first EVAL!

The halting problem

- If there is a problem that the universal machine can’t solve, then no machine can solve, and hence no effective process
- Make list of all possible programs (all machines with 1 input)
- Encode all their possible inputs as integers
- List their outputs for all possible inputs (as integer, error or loops forever)
- Define f(n) = output of machine n on input n, plus 1 if output is a number
- Define f(n) = 0 if machine n on input n is error or loops
- But f can’t be computed by any program in the list!
- Yet we just described process for computing f??
- Bug is that can’t tell if a machine will always halt and produce an answer

The Halting theorem

- Halting problem: Take as inputs the description of a machine M and a number n, and determine whether or not M will halt and produce an answer when given n as an input
- Halting theorem (Turing): There is no way to write a program (for any computer, in any language) that solves the halting problem.

Turing’s history

- Published this work as a student
- Got exactly two requests for reprints
- One from Alonzo Church (professor of logic at Princeton)
  - Had his own formalism for notion of an effective procedure, called the lambda calculus
- Completed Ph.D. with Church, articulating the Church-Turing Thesis:
  - Any procedure that could reasonably be considered to be an effective procedure can be carried out by a universal machine (and therefore by any universal machine)
- Worked as code breaker during WWII
- Key person in Ultra project, breaking German’s Enigma coding machine
- Designed and built the Bombe, machine for breaking messages from German Airforce
- Designed statistical methods for breaking messages from German Navy
- Spent considerable time determining counter measures for providing alternative sources of information so Germans wouldn’t know Enigma broken
- Designed general-purpose digital computer based on this work
- Turing test: argued that intelligence can be described by an effective procedure – foundation for AI
- World class marathoner – fifth in Olympic qualifying (2:46:03 – 10 minutes off Olympic pace)
- Working on computational biology – how nature “computes” biological forms.
- His death