6.001 SICP
Analyze Eval and Computability

• One more variant: Analyze evaluator
• Does every expression stand for a value?
• Are there things we can’t compute?
• Summary

Hi Prof. Grimson and Prof. Darrell,

So far, your class has a response rate of 29%. However, we're aiming to get at least 50% for each class. Please send out another email to your classlist, and stress how valuable their feedback is to both the instructors and future students of 6.001. Also, remind them that they can win VI Socks! The URL is [http://ug.mit.edu](http://ug.mit.edu)

Thanks for your cooperation!

Brian

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

• Dynamic vs. lexical scoping
• Applicative order vs. 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))))

in eval:
  ((assignment? exp) (eval-assignmt exp env))

(define (assignment? exp) (tagged-list? exp 'set!))
(define (eval-assignmt exp env)
  (set-variable-value! (assignmt-variable exp)
                       (eval (assignmt-value exp) env)
                       env))
```

• Sure be nice to avoid this cost.
Eval is expensive (and not very clever)

\[
\text{(define (fact n)} \\
\quad \text{(if (= n 1) 1 (* n (fact - n 1)))))}
\]

static analysis: work done before execution

- straight interpreter
  \[
  \text{expression} \rightarrow \text{interpreter} \rightarrow \text{value}
  \]
- advanced interpreter or compiler
  \[
  \text{expr} \rightarrow \text{static analysis} \rightarrow \text{execution} \rightarrow \text{value}
  \]

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

\[
\text{expr} \rightarrow \text{analyze} \rightarrow \text{EP} \rightarrow \text{execution} \rightarrow \text{value}
\]

\[
\text{analyze}:\ expression \rightarrow (\text{Env} \rightarrow \text{anytype})
\]

\[
\text{(define (analyze exp) (lambda (env) exp))}
\]

Implementing analyze

\[
\text{(define (analyze exp) (cond)
  ((number? exp) (analyze-number exp))
  ((variable? exp) (analyze-variable exp))
  ((assignment? exp) (analyze-assignmt exp))
  . . .
  ))}
\]

Implementing number analysis

\[
\text{• analyze-number is easy}
\]

\[
\text{(define (analyze-number exp) (lambda (env) exp))}
\]

\[
\text{(black: analysis phase) (blue: execution phase)}
\]

\[
\text{(define (a-eval exp env) ((analyze exp) env))}
\]
Analyzing definitions

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

  (definition-variable exp)  x
  (definition-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))))

Example of analyze: variable name lookup

Subexpressions

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

- analysis phase:
  (analyze 'if   )  ==> pproc
  (analyze (= )  )  ==> cproc
  (analyze ('   ) )  ==> 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)))))

Example of analyze: variable name lookup
Implementing lambda

- Body stored in double bubble is an execution procedure
- Old make-procedure
  - list<symbol>, expression, Env \rightarrow Procedure
- New make-procedure
  - list<symbol>, (Env\rightarrow anytype), Env \rightarrow Procedure

\[
\text{(define (analyze-lambda exp)
  (let ((vars (lambda-parameters exp))
        (bproc (analyze (lambda-body exp))))
  (lambda (env)
    (make-procedure vars bproc env)))}
\]

Implementing apply: analysis phase

\[
\text{(apply (foo n) (bar x) (baz y))}
\]

\[
\text{(define (analyze-application exp)
  (let ((fproc (analyze (operator exp)))
        (aprocs (map analyze (operands exp))))
    (lambda (env)
      (execute-application
       (fproc env)
       (map (lambda (aproc) (aproc env))
            aprocs)))))
\]

Implementing apply: execution phase

\[
\text{(define (execute-application proc args)
  (cond
    ((primitive-procedure? proc)
     ...
    ((compound-procedure? proc)
     ((procedure-body proc)
      (extend-environment (parameters proc) args
        (environment proc))))
    (else ...)))}
\]

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
  - Recursively call analyze on subexpressions
  - Create an execution procedure which stores the EPs for subexpressions as local state
  - Double bubble stores execution procedure, not expression

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

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

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)

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
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!
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?

Deep Question #1
Does every expression stand for a value?

Some Simple Procedures
• Consider the following procedures
  (define (return-seven) 7)
  (define (loop-forever) (loop-forever))
• So
  (return-seven) ⇒ 7
  (loop-forever) ⇒ [never returns!]
• Expression (loop-forever) does not stand for a value; not well defined.

Syntax and Semantics
• Syntax: structure
• Semantics: meaning
• In English
  • syntax: the structure of a sentence
    – The dark blue box fell quickly.
  • semantics: what that sentence means
• In Scheme
  (define (fact n) (if (= n 0) 1 (* n (fact (- n 1)))))
  • what value does (fact 10) produce?
• The colorless green ideas slept furiously.
• Are there syntactically valid but meaningless scheme programs?

Deep Question #2
Are there well-defined things that cannot be computed?

halts?
• Even our simple procedures can cause trouble. Suppose we wanted to check procedures before running them to catch accidental infinite loops.
• Assume a procedure halts? exists:
  (halts? p) ⇒ #t if (p) terminates
  ⇒ #f if (p) does not terminate
• halts? is well specified – has a clear value for its inputs
  (halts? return-seven) ⇒ #t
  (halts? loop-forever) ⇒ #f

The Halting Theorem:

Procedure halts? cannot exist. Too bad!

- Proof (informal): Assume halts? exists as specified.
  
  (define (contradict-halts)
    (if (halts? contradict-halts)
        (loop-forever)
        #t))

  (contradict-halts)
  ⇒ ?????

- Wow! If contradict-halts halts, then it loops forever.
- Contradiction!
  Assumption that halts? exists must be wrong.

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.

Some we can’t, but some we can…

So:

- There are some well specified things we cannot compute

But…

- There are also interesting things we CAN compute
  Many of our most interesting (powerful) procedures have been recursive

(1) Abstraction

- Elements of a Language
  
  - Procedural Abstraction:
    - Lambda – captures common patterns and "how to" knowledge
  
  - Functional programming & substitution model
  
  - Conventional interfaces:
    - list-oriented programming
    - higher order procedures

(2) Data, State and Objects

- Data Abstraction
  - Primitive, Compound, & Symbolic Data
  - Contracts, Abstract Data Types
  - Selectors, constructors, operators, ...

- Mutation: need for environment model

- Managing complexity
  - modularity
  - data directed programming
  - object oriented programming

(3) Language Design and Implementation

- Evaluation – meta-circular evaluator
  
  - eval & apply

- Language extensions & design
  - lazy evaluation
  - dynamic scoping
  - wild ideas, e.g. nondeterministic computation
(3) Language Design and Implementation

- Evaluation – meta-circular evaluator
  - eval & apply

- Language extensions & design
  - lazy evaluation
  - dynamic scoping
  - wild ideas, e.g. nondeterministic computation

Didn’t cover this term: Register machines

- ec-eval: evaluator implemented in machine code
- compilation: convert Scheme program to machine code
- implement and manage list structured data

Go forth and compute!

Hi Prof. Grimson and Prof. Darrell,

So far, your class has a response rate of 29%. However, we’re aiming to get at least 50% for each class. Please send out another email to your classlist, and stress how valuable their feedback is to both the instructors and future students of 6.001. Also, remind them that they can win VI Socks! The URL is http://ug.mit.edu

Thanks for your cooperation!

Brian