Inductive Specification and Recursion

Gul Agha
CS 4211/23/2006

Inductive Sets of Data

- Sets can be defined by induction
- Example: Even Numbers
  - S is the smallest set of natural numbers satisfying:
    1. \(0 \in S\)
    2. If \(x \in S\) then \(x + 2 \in S\)
  - Defines the set of numbers that are multiples of 2. (Proof by Induction)

Example: List of Numbers

The set \(\text{list-of-numbers}\) is the smallest set of values such that:
1. The empty list is a list-of-numbers
2. If \(l\) is a list of numbers and \(n\) is a number, then the pair \((n . l)\) is a list of numbers

Can apply the rules to show that the following are lists of numbers:

- \((\quad)\)
- \((14 . (\quad))\)
- \((3 . (14 . (\quad)))\)
- \((-7 . (3 . (14 . (\quad))))\)

Sets using BNF

1. \(<\text{list-of-numbers}> ::= (\quad)\)
2. \(<\text{list-of-numbers}> ::= (\langle\text{number}\rangle . <\text{list-of-numbers}>\)\)

How to prove a property about a list of numbers?
1. Prove true for empty list (base case)
2. Prove: if true for a list of length \(n\), also true for list of length \(n + 1\)

Induction

If we can show:
- Property \(P\) is true for all the base cases (e.g., all sentences in the language derived by a single application of the rules)
- Given property \(P\) is true for all sentences derived using \(n\) applications of the rules, we can prove the property is true for all sentences derived by using \(n + 1\) applications of the rules

then the property is true for all sentences of the language.

Course of Values Induction

- On natural numbers \(\forall m \in \omega . Q(m)\)
  1. Base case \(\forall k < 0 . Q(k)\)
  2. Induction \(\forall m \in \omega ((\forall k < m . Q(k)) \Rightarrow (\forall k < m + 1 . Q(k)))\)

But base case is vacuously true and 2. simplifies to showing:

\(\forall m \in \omega . (\forall k < m . Q(k) \Rightarrow Q(m))\)
Well-Founded Relation

- All these cases of induction are special cases of well-founded induction
- Well-founded induction is done over well-founded relations

Def. A well-founded relation is a binary relation < on a set A such that there are no infinite descending chains ... < a_i < ... < a_1 < a_0

i.e., every nonempty subset has a minimal element.

Well-Founded Induction

- The principle of well-founded induction:
  Let < be a well-founded relation on a set A.
  Let P be a property.
  \( \forall a \in A. P(a) \iff (\forall b < a. P(b)) \Rightarrow P(a) \)

Called Noetherian induction after the algebraist Noether.
(Computer Scientists mislabel it "structural induction").

Induction on Derivations

- Well-founded induction leads to induction on derivations of elements derived from grammars specified by a BNF.
- Length of shortest derivation can provide an ordering: notice that more productions cannot shrink elements.
- Recursive programs working on inductively specified data elements need to work on elements with longer derivations given that they can work on elements with shorter derivation.

Kleene Closures

\[
\text{Kleene star} \\
<\text{list-of-numbers}> ::= (\{\text{<number>}\}^*)
\]

\[
\text{Kleene plus} \\
<\text{list-of-numbers}> ::= (\{\text{<number>}\}^+) -- nonempty lists
\]

\[
\text{Separated list} \\
\{\text{<expression>}\}^*(c) -- sequence of any number of non-terminal <expression> separated by nonempty character c
\]

Recursion on derivations

- Functions on recursive datatypes (e.g., lists) tend to be recursive
- Recursion over recursive datatypes generally by structural recursion
  - Recursive calls made to components of structure of the same recursive type.
  - Base cases of recursive types stop the recursion of the function.
  - Proof of correctness follows by induction.

Forward Recursion

- Split input into components using the rules of the grammar
- In forward recursion, first call the function recursively on all the recursive components, and then built the final result for the partial results
- Wait until the whole structure has been traversed to start building the answer
Example of Recursively Specified Program

<s-list> \(\rightarrow\) (\(<\text{symbol-expression}>>\)*)

i.e.: 
<s-list> \(\rightarrow\) ( )
| (\(<\text{symbol-expression}>>\),<s-list>)

<symbol-expression> \(\rightarrow\) <symbol>
| <s-list>

Grammar contains two non-terminals (mutually recursive data specification).

Example of s-list

<s-list> \(\rightarrow\) (\(<\text{symbol-expression}>>\)*)
<symbol-expression> \(\rightarrow\) <symbol> | <s-list>

(a b c)
(a (a b) c)
((b c) (b ( ) d))
((b c) (b (a d) (c d e)) ( ) d)

Example:
Problem specification
Want to create a list which replaces all occurrences of a given element by another given element

\[
\text{(subst 'a 'b '((b c) (b ( ) d)))}
\]
\[
\text{((a c) (a ( ) d))}
\]

Follow the grammar

<s-list> \(\rightarrow\) ( ) | (<symbol-expression>,<s-list>)
(define subst
(lambda (new old slist)
  (if (null? slist)
    '( )
    ....)

<symbol-expression> \(\rightarrow\) <symbol> | <s-list>
(define subst-in-symbol-expression
(lambda (new old se)
  ....)

Follow the grammar

<s-list> \(\rightarrow\) (<symbol-expression>,<s-list>)

(define subst
(lambda (new old slist)
  (if (null? slist)
    '( )
    ....)

<symbol-expression> \(\rightarrow\) <symbol> | <s-list>

(define subst-in-symbol-expression
(lambda (new old se)
  ....)
Complexity

- Learn to break the rules.
- Don’t just follow the grammar into an intractable pit!

Common big-O times:
- Constant time $O(1)$
  - input size doesn’t matter
- Linear time $O(n)$
  - double input ⇒ double time
- Quadratic time $O(n^2)$
  - double input ⇒ quaduple time
- Exponential time $O(2^n)$
  - increment input ⇒ double time

How long will it take?

How long will it take?

Linear Time

- Expect most list operations to take linear time $O(n)$
- Each step of the recursion can be done in constant time
- Each step makes only one recursive call
- List example: *, list, append
- Integer example: factorial

Quadratic Time

- Each step of the recursion takes time proportional to input
- Each step of the recursion makes only one recursive call.
  > (define bad-rev
   (lambda (lis)
     (if (null? lis)
       lis
       (append (bad-rev (cdr lis))
               (list (car lis))))))

Exponential running time

- Hideous running times on input of any size
- Each step of recursion takes constant time
- Each recursion makes two recursive calls
- Easy to write naïve code that is exponential for functions that can be linear

> (bad-rev '(1 2 3))
| (bad-rev (1 2 3))
| (bad-rev (2 3))
| (bad-rev (3))
| (1)
| (3)
| (3 2)
| (3 2 1)
| (3 2 1)

> (bad-rev '(1 2 3 4))
| (bad-rev (1 2 3 4))
| (bad-rev (2 3 4))
| (bad-rev (3 4))
| (bad-rev (4))
| (4)
| (append (4) (3))
| (4 3)
| (4 3 2)
| (4 3 2 1)
| (4 3 2 1)
Exponential running time

- (define naiveFib
  (lambda (n)
    (cond ((= n 0) 0)
          ((= n 1) 1)
          (else (naiveFib (- n 1))
                (naiveFib (+ n 1))))))

Tail Recursion

- The result of the call is the result produced by the caller (no operations performed after the call).
- Example of tail recursion: While loops, do loops
- In Scheme, all tail recursion is optimized:
  - Same efficiency as a while loop
  - Interpreter/Compiler converts recursion into tail recursive form through continuation-passing style transform.
  - Don’t worry about using tail recursion – focus on complexity!

Example..

- Recursive Factorial

  (define factorial
    (lambda (n)
      (if (= n 0)
          1
          (* n (factorial (- n 1))))))

Tracing a recursive factorial

[(factorial 5)
 (factorial 4)
 (factorial 3)
 (factorial 2)
 (factorial 1)
 (factorial 0)
 1
 2
 6
24
120
120]

Iterative Factorial

- (define factorial2
  (lambda (n)
    (fact-itr n 1)))

Tracing Iterative Factorial

> (factorial2 5)
  (fact-itr 5 1)
  (fact-itr 4 5)
  (fact-itr 3 20)
  (fact-itr 2 60)
  (fact-itr 1 120)
  (fact-itr 0 120)
  120
  120
The Variety of Local Definitions

- Should not use global (Define foo ...) when foo is needed only locally
- The order of bindings in let is unspecified
- The order of bindings in let* is left to right (the order of lexical specification)
- Mutually recursive bindings can be specified by letrec

Example of Mutually Recursion

> (letrec ((even? (lambda (n) (if (zero? n) #t (odd? (- n 1)))))
  (odd? (lambda (n) (if (zero? n) #f (even? (- n 1)))))
  (even? 88))
#t

The Varieties of Equivalence

- procedure: eqv? obj1 obj2
  - The `eqv?' procedure defines a useful equivalence relation on objects. Briefly, it returns #t if obj1 and obj2 should normally be regarded as the same object.
  - Examples:
    - string=?
    - both #t or #f
    - equal numbers
    - Symbols with the same strings
    - ...

Examples of eqv

(eqv? 'a 'a) ==> #t
(eqv? 'a 'b) ==> #f
(eqv? 2 2) ==> #t
(eqv? '(') ')') ==> #t
(eqv? 100000000 100000000) ==> #t
(eqv? (cons 1 2) (cons 1 2)) ==> #f
(eqv? (lambda () 1) (lambda () 2)) ==> #f
(eqv? '#f 'nil) ==> #f
(let ((p (lambda (x) x))) (eqv? p p)) ==> #t

Equivalence predicates (contd.)

(eqv? 'a 'a) ==> unspecified
(eqv? '#() '#()) ==> unspecified
(eqv? (lambda (x) x) (lambda (x) x)) ==> unspecified
(eqv? (lambda (x) x) (lambda (y) y)) ==> unspecified
(eqv? '(a) '(a)) ==> unspecified
(eqv? "a" "a") ==> unspecified
(eqv? '(b) (cdr '(a b))) ==> unspecified
(let ((x '(a))) (eqv? x x)) ==> #t

A Finer Equivalence

- `eq?' is similar to `eqv?' except ...
- `eq?' may detect finer distinctions `eqv?'.
- `eq?' and `eqv?' are guaranteed to have the same behavior on symbols, booleans, the empty list, pairs, procedures, and non-empty strings and vectors.
- `eq?'s behavior on numbers and characters is implementation-dependent, but it will always return either true or false, and will return true only when `eqv?' would also return true.
- `Eq?' may also behave differently from `eqv?' on empty vectors and empty strings.
Equal

- **library procedure**: `equal? obj1 obj2`
  - `equal?` recursively compares the contents of pairs, vectors, and strings, applying `eqv?` on other objects such as numbers and symbols.
  - A rule of thumb is that objects are generally `equal?` if they print the same.
  - `equal?` may fail to terminate if its arguments are circular data structures.

Examples of equal

- `(equal? 'a 'a) ==> #t`
- `(equal? 'a (a)) ==> #t`
- `(equal? (a b) c '(a (b) c)) ==> #t`
- `(equal? "abc" "abc") ==> #t`
- `(equal? 2 2) ==> #t`
- `(equal? (make-vector 5 'a) (make-vector 5 'a)) ==> #t`
- `(equal? (lambda (x) x) (lambda (y) y)) ==> unspecified`