Structural Induction - application of induction

Inductive definition

An **inductive definition** of a concept consists of:

- **foundation rule** = where are simplest instances of the concept.
- **constructor rule** = how to combine simpler instances of a concept into a more complex instance

Example: well-formed boolean expression

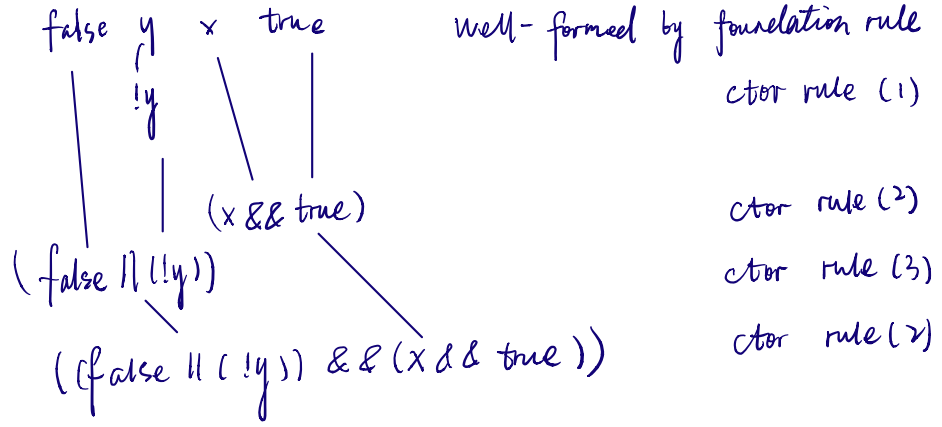
A **boolean expression** consists of combinations of: `true`, `false`, `&&`, `||`, `!`, and boolean variables (such as x or y, where these have been declared to be boolean variables)

A **well-formed boolean expression** (WFBE) can be defined as follows:

- (foundation rule) `true`, `false`, and x (where x is a boolean variable) are well-formed
- (constructor rule) if A and B are well-formed, then so are: `( !A )` (1), `( A && B )` (2), `( A || B )` (3)

ctor

Using the definition above, show that `( ( false || (!y ) ) && ( x && true ) )` is well-formed



Proof by structural induction

Prove : Every well-formed boolean expression contains an equal number of left and right parentheses.

Proof by structural induction :

P(*n*) : Every WFBE built using *n* applications of the constructor rule contains an equal number of L and R parenthesis.

Base case : Show P(0) holds.

0 applications of ctor rule means WFBE was created using just foundation rule

Case : true

Case : false

Case : x (some boolean variable).

} 0 parens in all cases

so # L parens = # R parens = 0

Inductive step : Assume P(*j*) holds for 0 ≤ *j* ≤ *k*. Show P(*k*+1) holds. Strong Induction

Induction hypothesis : Every WFBE built using *j* applications of the constructor rule, where 0 ≤ *j* ≤ *k*, contains an equal number of L and R parentheses.

For any WFBE E, let L_E = # L parens in E, R_E = # parens in E.

Let S be WFBE created using k+1 applications of the ctor rule.

Consider the $(k+1)^{st}$ application.

Case : Used rule `( !A )` (where `A` is a WFBE)

A

|

!A ← S

A was formed by application of ctor rule.

By (IH) $L_A = R_A$.

$L_S = 1 + L_A$. → Since $L_A = R_A$, these 2 are equal, i.e. $L_S = R_S$.

$R_S = 1 + R_A$

Case : Used rule `( A && B )` (where `A` and `B` are WFBE)

A B

\ /

(A && B)

↑

S

ctor apps to get A is between 0 and k, inclusive.

- - - - - - B - - - - - - - - - - -

By (IH) $L_A = R_A$, $L_B = R_B$.

$L_S = 1 + L_A + L_B$ so these sums are the same, i.e. $L_S = R_S$.

$R_S = 1 + R_A + R_B$.

Case : Used rule `( A || B )` (where `A` and `B` are WFBE)

By same argument as for && case, $L_S = R_S$

Regardless of which part of ctor rule was used in $(k+1)^{st}$ application, $L_S = R_S$.

So P(k+1) holds. □

∴ by strong induction, P(n) holds ∀ n ∈ ℕ. ■

Recursion

Terms

recursive definition = definition of a term or concept that includes that terms/concept in the definition.

Examples: GNU = Gnu's not Unix.

Virahanka numbers = 1, 1, 2, 3, 5, ---- .

$V_1 = 1$, $V_2 = 1$, $V_n = V_{n-1} + V_{n-2}$ for $n > 2$

- inductive definitions are recursive definitions.

See also Infinite cat project.

recursive algorithm solve a problem by reducing it to (one or more) instances of the same problem with smaller input.

Example: recursive powering problem

Input: $a \in \mathbf{Z}$, $b \in \mathbf{Z}^+$

Output: $a^b = a \cdot a \cdot \ldots \cdot a$ (i.e., a multiplied b times)

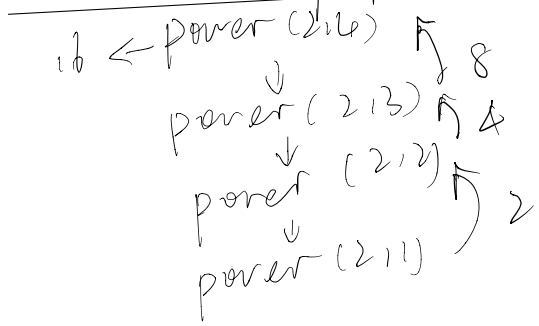
Algorithm:

procedure *power*(*a*, *b*)

(1) if $b = 1$ then return a

(2) else return $a \cdot$power(a, b - 1)

Trace execution of power(2, 4)



Proving program correctness of recursive algorithms

Partial correctness correct result is returned in non-recursive case (base case)

Show that for all valid inputs x

- valid input is given to all recursive calls (if any) made by the program with input x
- assuming the recursive calls return the correct output (for their input) and the program terminates, the program produces the correct output for input x

recursive call. or recursive step.

Termination

Show that the chain of recursive calls must eventually end.

- prove by induction on some quantity that decreases with each recursive call.

related to input values.

heading towards base case values.

Program correctness: recursive powering algorithm

Partial correctness: Assume input is valid, i.e. $a \in \mathbb{Z}$, $b \in \mathbb{Z}^+$

case *b* = 1 : base case.
algorithm returns a, which is correct for $a^b = a^1$

case *b* ≠ 1 : (recursive call)
algorithm returns a · power(a, b-1)
need to show valid input for ↰
we have assumed $a \in \mathbb{Z}$
since $b \in \mathbb{Z}^+$ and $b \neq 1$, $b > 1$
Then $b - 1 > 0$ so $(b-1) \in \mathbb{Z}^+$
Assume recursive call returning the correct output (& terminates).
(show orginal call, power(a, b) returns correct output)
So power(a, b-1) returns a^{b-1}.
The algorithm returns $a \cdot a^{b-1} = a^b$, which is correct result.

∴ in all cases with valid input, power(a, b) produces the correct result.

Termination:

Prove by induction on b that program terminates for all $a \in \mathbb{Z}$, $b \in \mathbb{Z}^+$.

P(*n*) : Program terminates on input a, n, where $a \in \mathbb{Z}$, $n \in \mathbb{Z}^+$.
i.e. power(a, n) terminates.

Base case : show P(1) holds.
when n = b = 1, program terminates right away (line 1)

Inductive step : show P(k) ⇒ P(k+1)

Induction hypothesis : power(a, k) terminates (IH).

Consider power(a, k+1)
Since k+1 > 1, algorithm call power(a, k) & then multiply result by a (line 2)
By IH, power(a, k) terminates, multiplying by a takes 1 step.
∴ power(a, k+1) terminates

∴ by induction, power(a, n) terminates for all $a \in \mathbb{Z}$, $n \in \mathbb{Z}^+$. ■

Recursion (continued)

Example: recursive GCD

Input: $a, b \in \mathbf{Z}^+$
Output: gcd(a, b), i.e., the greatest common divisor of a and b

Algorithm:

procedure *GCD*(a, b)
(1) if $a = b$ then return a
(2) if $a < b$ then return *GCD*(a, $b - a$)
(3) if $a > b$ then return *GCD*($a - b$, b)

tail recursion
when rec call is last thing
procedure does before it returns
(& return value is value of recursive call

Trace
b ← GCD(48, 30)
GCD(18, 30) b
GCD(18, 12) b
GCD(6, 12) b
GCD(6, 6) b

Program correctness: recursive GCD algorithm

Show that the recursive *GCD* algorithm correctly implements the program specification by establishing (a) partial correctness and (b) termination.

Partial correctness Assume input is valid, i.e. $a, b \in \mathbb{Z}^+ = \mathbb{N}^+$, and program terminates.

Case $a = b$
algorithm return a ← which is correct gcd(a, a) = a.

Case $a < b$
algorithm return GCD(a, b-a) recursive call.
show valid input given to GCD(a, b-a)
we know $a \in \mathbb{Z}^+$, since $b > a$, $b - a > 0$, so $b - a \in \mathbb{Z}^+$
Since input to recursive call is valid, we assume the recursive call returns the correct output, namely gcd(a, b-a)
Program returns gcd(a, b-a), by previous Lemma, when $a < b$, gcd(a, b) = gcd(a, b-a)
So program returns gcd(a, b) when is correct for GCD

Case $a > b$
note that gcd(b, a) = gcd(a, b)
Argument is same as case $a < b$ but with a & b reversed.

Termination

Consider *a* + *b*

Prove by induction on the sum *a* + *b* that the program terminates.

P(*n*) : program terminate when a+b = n

Base case : Show program terminates on the smallest sum *a* + *b* for valid input *a* and *b*

Since a,b ∈ Z+, the smallest sum is a+b=2
Then a=b=1, so algorithm terminate on line 1 now

Inductive step :

Case a=b, algorithm terminate
IH: If 2 ≤ a+b ≤ k, and a,b ∈ Z+, the algorithm holds
strong: show algo terminates when a+b = k+1, Algorithm call GCD(x*, y*) terminate
Using same arguments partial correct ness, we know the recursive call's valid input i.e. a*, b* ∈ Z+, so a*+b* ≥ 2.

$a^* + b^* = a+b-1 = k+1-1 = k$ $(a^*+b^* \le (a+b-1))$

Either $a^* = a$, $b^* = b-a = b-1$.

or $a^* = b-a$, $b^* = a$. $a^*+b^* = b+a$

Either way, $a^*+b^* \le a+b-1 = k$.

By IH, GCD(a*, b*) terminates

Thus, algorithm, which GCD(a*, b*) & return that result also terminates

∴ By strong induction, algorithm terminates on valid input.