EDAA40
Discrete Structures in Computer Science

6: Induction and recursion

Jörn W. Janneck, Dept. of Computer Science, Lund University

introduction

$$f(n) = \begin{cases} 1 & \text{for } n = 0 \\ n \cdot f(n-1) & \text{otherwise} \end{cases}$$

$$g(n) = \begin{cases} 1 & \text{for } n \leq 1 \\ g(n-2) + g(n-1) & \text{otherwise} \end{cases}$$

introduction

Wilhelm Ackermann
1896-1942

$$A(m,n) = \begin{cases} n+1 & \text{for } m = 0 \\ A(m-1,1) & \text{for } m > 0, n = 0 \\ A(m, A(m, n-1)) & \text{otherwise} \end{cases}$$

$$A(m,n) = \begin{cases} n+1 & \text{for } m = 0 \\ A(m-1,1) & \text{for } m > 0, n = 0 \\ A(m-1, A(m, n-1)) & \text{otherwise} \end{cases}$$

induction and recursion

Induction and recursion:

defining sets, and especially functions **(recursion)**

proving properties of things **(induction)**

a simple inductive proof

$\boxed{\mathrm{sum}(n) = \sum_{i=1...n} i = 1 + ... + n}$

Hypothesis: $\mathrm{sum}(n) = \frac{n(n+1)}{2}$

Basis: $\mathrm{sum}(1) = \frac{1(1+1)}{2} = 1$

Induction step:

Assuming that $\mathrm{sum}(k) = \overbrace{\frac{k(k+1)}{2}}^{\text{induction hypothesis}}$

show that $\underbrace{\mathrm{sum}(k+1) = \frac{(k+1)(k+2)}{2}}_{\text{induction goal}}$

$$\begin{aligned} \mathrm{sum(k+1)} &= \mathrm{sum(k)} + \mathrm{(k+1)} && \text{def sum()} \\ &= \frac{k(k+1)}{2} + (k+1) && \text{IH} \\ &= \frac{k(k+1)+2(k+1)}{2} \\ &= \frac{(k+1)(k+2)}{2} \end{aligned}$$

a simple induction principle

Hypothesis: $P(n)$ for all n

Basis: Show that $P(1)$ (or $P(0)$ or some other, depending on circumstances)

Induction step: Assuming that $\underbrace{P(k)}_{\text{induction hypothesis}}$ show that $\underbrace{P(k+1)}_{\text{induction goal}}$

Things that often go wrong:

- mixing basis and induction step: do not try to do everything at once
- confusing induction hypothesis and induction goal
- not using the induction hypothesis: it ain't cheating!
- getting lost: proving the goal can be messy, keep eyes on prize

defining large (infinite) sets

Remember these from the first lecture?

recursive definition (we will discuss this later now)

enumeration w/ suspension points/ellipsis

$\{1, 2, 3, 4, 5, \ldots\}$

(informal stand-in for a recursive definition)

Many of these infinite sets are *functions*!

simple recursive definitions

$$f(n) = \begin{cases} 1 & \text{for } n = 0 \\ n \cdot f(n-1) & \text{otherwise} \end{cases}$$

Technically, this describes the following set:

$$f = \{(0, 1), (1, 1), (2, 2), (3, 6), (4, 24), (5, 120), \ldots\}$$

We could define this set also in this manner:

Do you see a more general principle?

$$f_0 = \{(0, 1)\}$$

$$f_{k+1} = f_k \cup \{(n, n \cdot v) : (n-1, v) \in f_k\}$$

$$f = \bigcup_{k \in \mathbb{N}} f_k$$

We could use more (even <u>all</u>) previous values here!

cumulative recursive definitions

$$g(n) = \begin{cases} 1 & \text{for } n \leq 1 \\ g(n-1) + g(n-2) & \text{otherwise} \end{cases}$$

Technically, this describes the following set:

$$g = \{(0, 1), (1, 1), (2, 2), (3, 3), (4, 5), (5, 8), (6, 13), (7, 21), \ldots\}$$

We could define this set also in this manner:

$$g_0 = \{(0, 1), (1, 1)\}$$

$$g_{k+1} = g_k \cup \{(n, v + w) : (n-1, v), (n-2, w) \in g_k\}$$

$$g = \bigcup_{k \in \mathbb{N}} g_k$$

Cumulative recursive definitions reach back further than the last defined value, possibly to all previously defined values!

cumulative (complete) induction

Cumulative, also *complete* or *strong*, induction uses an induction hypothesis that assumed the truth of the hypothesis for <u>all</u> smaller values, instead of just the previous one.

$$g(n) = \begin{cases} 1 & \text{for } n \leq 1 \\ g(n-1) + g(n-2) & \text{otherwise} \end{cases}$$

Hypothesis: $g(n) \leq 2^n$

Basis: $g(0) \leq 2^0 = 1$

$g(1) \leq 2^1 = 2$

Induction step:

$\boxed{\text{Assuming that } \overbrace{g(m) \leq 2^m \text{ for all } m < k}^{\text{induction hypothesis}} \quad \text{show that } \underbrace{g(k) \leq 2^k}_{\text{induction goal}}}$

$g(k) = g(k-1) + g(k-2)$ def g()

$\leq 2^{k-1} + 2^{k-2}$ IH

$= \frac{1}{2}2^k + \frac{1}{4}2^k$

$\leq 2^k$

cumulative induction principle

Hypothesis: $P(n)$

Basis: $P(0), \ldots$

Induction step:

Assuming that $\underbrace{P(m) \text{ for all } m < k}_{\text{induction hypothesis}}$ show that $\underbrace{P(k)}_{\text{induction goal}}$

Note that the basis is the vacuous form of the induction step, for k=0.
As a result, the basis is subsumed by the induction step.
In practice, it is often treated separately.

closure under relation

Given a relation $R \subseteq A \times A$ and a set $X \subseteq A$, the *closure of X under R* $R[X]$ is defined as the smallest $Y \subseteq A$ such that

$X \subseteq Y$ and $R(Y) \subseteq Y$

Construction:

$Y_0 = X$

$Y_{n+1} = Y_n \cup R(Y_n)$

$R[X] = \bigcup_{i \in \mathbb{N}} Y_i$

When R is understood, we also write X^+.

C^+ ?

$\{\varepsilon\}^+$?

What is the set of all palindromes?

Section 2.7.2 in SLAM.

Example:

$C = \{"a", ..., "z"\}$

$R \subseteq C^* \times C^*$

$R = \{(s, asa) : s \in C^*, a \in C\}$

closure of relations, rules, generators

Given a set A, a family of relations on A $R = \{R_i \subseteq A^{n_i} : i \in I\}$ and a set $X \subseteq A$, the *closure of X under R* $R[X]$ is defined as the smallest $Y \subseteq A$ such that

$$X \subseteq Y \text{ and } R_i(Y^{n_i-1}) \subseteq Y \text{ for all } i \in I$$

Construction:

$$Y_0 = X$$

$$Y_{n+1} = Y_n \cup \bigcup_{i \in I} R_i(Y_n^{n_i-1})$$

$$R[X] = \bigcup_{i \in \mathbb{N}} Y_i$$

When R is understood, we also write X^+.

The elements of R are also called *rules, constructors, generators.*

Example:

$$R = \{R_1, R_2, R_3\}, C = \text{UTF-16}$$

$$R_1 = \{(s, \text{"-"}\ s) : s \in C^*\}$$

$$R_2 = \{(s_1, s_2, \text{"("}\ s_1\ \text{"+"}\ s_2\ \text{")"}) : s_1, s_2 \in C^*\}$$

$$R_3 = \{(s_1, s_2, \text{"("}\ s_1\ \text{"*"}\ s_2\ \text{")"}) : s_1, s_2 \in C^*\}$$

$$V = \{\text{"a"}, ..., \text{"z"}\}^* \setminus \{\varepsilon\}$$

V^+ ?

examples of closures in CS

Syntax of statements in C

Statements in C are defined using extended BNF as follows.

```
<stmt> ::= ;
        | <exp>;
        | { <stmt-list> }
        | if (<exp>)  <stmt>  | if (<exp>) <stmt> else <stmt>
        | while ( <exp> ) <stmt>
        | do <stmt> while (<exp>) ;
        | for (<opt-exp>; <opt-exp>; <opt-exp>) <stmt>
        | switch ( <exp> ) <stmt>
        | case <const-exp> : <stmt>
        | default : <stmt>
        | break;   | continue;   | return;   | return <exp>;
        | goto <label>;  |  <label> : <stmt>
<stmt-list> ::= <stmt> *
<opt-exp> ::= ε | <exp>
```

```
public abstract class List<T> {...
  public abstract List<T> Append(List<T> that);
  public abstract List<U> Flatten<U>();
}
public class Nil<A> : List<A> {...
  public override List<U> Flatten<U>()
    { return new Nil<U>(); }
}
public class Cons<A> : List<A> {...
  A head; List<A> tail;
  public override List<U> Flatten<U>()
  { Cons<List<U>> This = (Cons<List<U>>) (object) this;
    return This.head.Append(This.tail.Flatten<U>()); }
}
```

structural induction

$$R = \{R_1, R_2, R_3\}, C = \text{UTF-16}$$

$$V = \{\text{"a"}, ..., \text{"z"}\}^* \setminus \{\varepsilon\}$$

$$R_1 = \{(s, \text{"-"}\ s) : s \in C^*\}$$

$$R_2 = \{(s_1, s_2, \text{"("}\ s_1\ \text{"+"}\ s_2\ \text{")"}) : s_1, s_2 \in C^*\}$$

$$R_3 = \{(s_1, s_2, \text{"("}\ s_1\ \text{"*"}\ s_2\ \text{")"}) : s_1, s_2 \in C^*\}$$

Hypothesis: Every $s \in R[V]$ contains equal numbers of opening and closing parentheses. ✓

Basis: Every $s \in V$ contains equal numbers of opening and closing parentheses. ✓

Induction step:

induction hypothesis

All rules $R_i \in R$ preserve the property: if their "input" objects have it, then so does their "output" object.

induction goal

$$R_1 = \{(s, \text{"-"}\ s) : s \in C^*\}$$

$$R_2 = \{(s_1, s_2, \text{"("}\ s_1\ \text{"+"}\ s_2\ \text{")"}) : s_1, s_2 \in C^*\}$$

$$R_3 = \{(s_1, s_2, \text{"("}\ s_1\ \text{"*"}\ s_2\ \text{")"}) : s_1, s_2 \in C^*\}$$

✓

structural induction principle

Structural induction is a variant of cumulative induction, but instead of natural numbers we induce over the structure of rule-generated objects in some structurally-recursive set $R[X]$.

Hypothesis: $P(x)$ for all $x \in R[X]$

Basis: $P(x)$ for all $x \in X$

Induction step:

induction hypothesis

All rules $R_i \in R$ preserve the property: if their "input" objects have it, then so does their "output" object.

induction goal

structural recursion on domains

$R = \{R_1, R_2, R_3\}, C = \text{UTF-16}$

$V = \{\text{"a"}, ..., \text{"z"}\}^* \setminus \{\varepsilon\}$

$R_1 = \{(s, \text{"-"}\ s) : s \in C^*\}$

$R_2 = \{(s_1, s_2, \text{"("}\ s_1\ \text{"+"}\ s_2\ \text{")"}) : s_1, s_2 \in C^*\}$

$R_3 = \{(s_1, s_2, \text{"("}\ s_1\ \text{"*"}\ s_2\ \text{")"}) : s_1, s_2 \in C^*\}$

Let's write a function that evaluates an expression in $R[V]$.

Assume a function $E : V \longrightarrow \mathbb{R}$ that assigns every variable a value.

$\mathrm{eval}_E : R[V] \longrightarrow \mathbb{R}$

$$\mathrm{eval}_E : s \mapsto \begin{cases} E(s) & \text{for } s \in V \\ -\mathrm{eval}_E(s') & \text{for } s' \in R[V], s = \text{"-"} s' \\ \mathrm{eval}_E(s_1) + \mathrm{eval}_E(s_2) & \text{for } s_1, s_2 \in R[V], s = \text{"("}\ s_1\ \text{"+"}\ s_2\ \text{")"} \\ \mathrm{eval}_E(s_1) \cdot \mathrm{eval}_E(s_2) & \text{for } s_1, s_2 \in R[V], s = \text{"("}\ s_1\ \text{"*"}\ s_2\ \text{")"} \end{cases}$$

Note how the structure is *decomposed* (or *deconstructed*) in these clauses!

Write a function that returns for every expression s the set of variables that occur in it.

unique decomposability

$R = \{R_1, R_2\}, C = \text{UTF-16}$

$V = \{\text{"a"}, ..., \text{"z"}\}^* \setminus \{\varepsilon\}$

$R_1 = \{(s_1, s_2, s_1\ \text{"+"}\ s_2) : s_1, s_2 \in C^*\}$

$R_2 = \{(s_1, s_2, s_1\ \text{"-"}\ s_2) : s_1, s_2 \in C^*\}$

Let's look at a variant of the previous example:

$$\mathrm{eval}_E : s \mapsto \begin{cases} E(s) & \text{for } s \in V \\ \mathrm{eval}_E(s_1) + \mathrm{eval}_E(s_2) & \text{for } s_1, s_2 \in R[V], s = s_1\ \text{"+"}\ s_2 \\ \mathrm{eval}_E(s_1) - \mathrm{eval}_E(s_2) & \text{for } s_1, s_2 \in R[V], s = s_1\ \text{"-"}\ s_2 \end{cases}$$

What is the problem with this function definition?

unique decomposability

Suppose we have generators $R = \{R_1, ..., R_k\}$ with $R_k \subseteq A^{n_k+1}$, and basis $X \subseteq A$.

The general form of a recursive function on $R[X]$ is

$$\mathrm{f} : x \mapsto \begin{cases} h_X(x) & \text{for } x \in X \\ h_1(f(x_1), ..., f(x_{n_1})) & \text{for } (x_1, ..., x_{n_1}, x) \in R_1 \\ & ... \\ h_k(f(x_1), ..., f(x_{n_k})) & \text{for } (x_1, ..., x_{n_k}, x) \in R_k \end{cases}$$

It is only well-defined if all $x \in R[X]$ are *uniquely decomposable.*

This means that for every x exactly one clause in the function definition must apply, and it must apply uniquely. This leads to two conditions for all $x \in R[X]$:

1. X and $R_i(R[X]^{n_i})$ pairwise disjoint
2. for all $x \in R_i(R[X]^{n_i}) : \#(R_i^{-1}(x)) = 1$

Compare to def. on page 99 in SLAM.

Match the general form above against the function definitions on the previous two slides.
Make sure you identify the problem on the previous slide.

we still haven't talked about...

$$A(m,n) = \begin{cases} n+1 & \text{for } m = 0 \\ A(m-1, 1) & \text{for } m > 0, n = 0 \\ A(m, A(m, n-1)) & \text{otherwise} \end{cases}$$

$$A(m,n) = \begin{cases} n+1 & \text{for } m = 0 \\ A(m-1, 1) & \text{for } m > 0, n = 0 \\ A(m-1, A(m, n-1)) & \text{otherwise} \end{cases}$$

Why is this case more difficult than factorial or Fibonacci?

well-founded sets

A poset $(A, <)$ is *well-founded* iff all non-empty subsets $X \subseteq A$ have a minimal element, i.e.

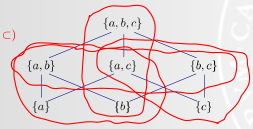
for some $m \in X$ and all $a \in X, a \not< m$

Intuitively, this means there are no infinite descending chains:

$$... < a_n < a_{n-1} < ... < a_2 < a_1 < a_0$$

minimal element != minimum element:
- minimal means that there is no smaller element
- minimum means all other elements are greater

$(\mathcal{P}(\{a,b,c\}) \setminus \{\emptyset\}, \subset)$



$(\mathbb{N}, <)$
$(\mathbb{Z}, <)$
$(\mathbb{Q}_0^+, <)$
$(\mathcal{P}(\mathbb{N}), \subset)$
$(\mathbb{N}^2, <_{\text{lex}})$
$(\mathbb{N}^2, <_{\text{prod}})$

well-founded induction

Emmy Noether
1882-1935

Cumulative (complete, strong) induction assumed that a property needed to be shown over the natural numbers.
Well-founded induction generalizes the idea to all well-founded sets.

Given a well-founded set $(A,<)$ and a property $P(a), a \in A$, if

$$\text{for all } a \in A : \underbrace{P(w) \text{ for all } w < a}_{\text{induction hypothesis}} \text{ implies } \underbrace{P(a)}_{\text{induction goal}}$$

then $P(a)$ for all $a \in A$

As with cumulative induction, the base case is subsumed by the induction step.
In practice, it is still often handled separately.

well-founded induction

Hypothesis: Every $n \geq 2$ can be factored into primes.

well-founded order: $(\{n \in \mathbb{N} : n \geq 2\}, |_{\neq})$ (divides-relation, strict version)

Descending chains?
Minimal elements?

(Base:) Trivially true for every minimal element in $(\{n \in \mathbb{N} : n \geq 2\}, |_{\neq})$.

Induction step: If n is not minimal, then there is a k such that $k|_{\neq}n$, and thus there is some m such that $n = km$. k and m can be prime-factored, therefore so can n.

How do we know that m can be factored into primes?

well-founded recursion

Given a well-founded set $(W,<)$ and a recursive function definition for a function $f : W \longrightarrow X$,
f is *well-defined* if it computes the value for every $w \in W$ only depending on values of f for $v < w$.

So how do we use this in practice? How can we tell that the Ackermann function is well-defined?

$$A : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$$

$$A(m,n) = \begin{cases} n+1 & \text{for } m = 0 \\ A(m-1,1) & \text{for } m > 0, n = 0 \\ A(m-1, A(m,n-1)) & \text{otherwise} \end{cases}$$

We need a well-founding of $\mathbb{N} \times \mathbb{N}$ such that

$$(m-1,1) \prec (m,0)$$

$$\begin{matrix}(m-1,x) \\ (m,n-1)\end{matrix} \prec (m,n)$$

Which could that be?
How does the erroneous Ackermann definition fail to be well-defined?