Project: Three types of projects

Ideal: related to your area of interest, helpful in your thesis / disc.

Possible: take a paper & present it.

Alternative: Do some research in theory of computing.

All other classes:
- given a problem
- you are expected to write a code for solving this problem

1st stage: any code

2nd stage: efficient code

3rd stage: problem is not- (SE) originally precisely formulated how do we know?

If original requirements (specs) cannot be satisfied, then we need to discuss with a customer which specs can be relaxed.

Intuition, experience.

Imp. problems:
- what can be computed is what cannot be computed?
- if it can be computed, then how? & on what devices it can be computed? & how much resources?
- if it cannot be computed, how can we relax the problem to make it computable?

What can be computed?

Equivalent form: what can be computed on any comp. device is:

Example: write in Java

As: we have a precisely formulated problem, we want to check whether this problem can be solved by a Java program?

Problem: Java is difficult to describe, so it is difficult to prove that something cannot be computed by a Java program.
what we need: Simplify the notion of Java-computable function.

Trajectory of the class: (NP hardness) toughest
what can be computed?
* standard computers, Java
* we can use other devices.

Alonzo Church: Church's Thesis: Anything that can be computed on any computational device can be also computed by a Java program.

Alan Turing: Turing Machine, 1936

Church-Turing Thesis:

R. Grandy 1970: much devices—Church's thesis is true.

As asserted: If quantum is used or electromagnetic church thesis is a statement about the physical world.

If church's thesis is correct—then to prove that something is not computable on any physical device, we need to prove that it is not computable by a Java program.

Let's start simplifying:
we have: * objects, * operations (+,-, ...)
* if-then statements
* loops (for, while, ...) many restrictions will be done
* other constructions

* Objects in Java: simplest: natural numbers 0,1,2,3...
* Strings, Integers, double, arrays, characters

Original: obj : Integers.

To integers in a comp, everything is a sequence of 0's.
$$a_0 = 1, \quad a_{n+1} = a^n + a$$

\text{CS description}

\begin{align*}
\text{for } (i = 0; i < n; i++) \\
\text{power} & = 1 \\
\text{power} & = \text{power} \times a \\
\text{power} (a, o) & = 1 \\
\text{for } (i = 0; i < n; i++) \\
\text{power} (a, i) & = \text{power} (\text{power} (a, i-1), a) \\
\end{align*}

\text{expression}

$$f = g$$

\text{before loop} \rightarrow \text{initialization.} \\
\text{for } (i = 0; i < n; i++) \\
\text{expression depending on } f \text{ (previous value), } a, \text{ and } i.

\text{parameters don't change during computation, hence } a_i^{m_i}

\text{f (m_1, \ldots, m_k, m) most of the time}

\begin{align*}
\text{f (m_1, \ldots, m_k, o) } & = g (m_1, \ldots, m_k) \\
\text{f (m_1, \ldots, m_k, m+1) } & = h (m_1, \ldots, m_k, f (m_1, \ldots, m_k, m))
\end{align*}

A. Church

\begin{align*}
\text{power : } k = 1 \\
\text{f (m, o) } & = 1 \\
\text{f (m, m+1) } & = f (m, m) \times m + 1 \\
\text{h (m, f, o)} & = m + f \\
\text{fact (0) } & = 1 \\
\text{fact (i+1) } & = \text{fact (i) } \times (i+1) \\
\text{f (m+1) } & = (m+1) \times f (m) \\
\text{h (m, f) } & = (m+1) \times f
\end{align*}

Minor Notations

\begin{align*}
\text{++, ---, +=, -=, \#} \\
\text{next, previous } f = PR (g, h) \\
\text{\Sigma } \text{Sum}
\end{align*}
Def (version 0):

A primitive recursive (P.R.) function is a function that can be obtained from \(0, 1, 2, \ldots\) and identity \(\text{Id}\) in \(\Pi^n_k\) by using composition & P.R.

Projection: \(f(n_1, \ldots, n_k) = n_i\) by using \(\Pi^1_k\).

\(\Pi^1_k\) is injective. If we draw a line from this point on \(n_i\) axis, it gives \(\text{3}\) of projection.

Q3: how can we further simplify this definition?

0 = \text{probably stays}

1 = 6(0)

2 = 6(6(0))

Conclusion: to describe which functions are P.R.

3 = 6(6(6(0)))

it is sufficient to keep only one constant: \(0\)

Caution: we are not talking about efficiency, size grows exponentially.

Unary notation for integers: \(5_{10} = 111_1\), \(10_{10} = 1010_1\)

Binary notation: \(10101_2\)

\(\Pi^1_k\) (Identity is projecting a special case)

\(\Pi^1_k\) is a projection, \(\Pi^1_k(n_1, \ldots, n_k) = n_i\)

Identify in terms of projection: \(\Pi^1_k(\text{n}) = \text{n}\)

Addition: do we need to it as basic operation?

\[a + b = a + 1 + \ldots + 1 \quad \text{\(b\) times} \quad \text{for } \(i = 0, i < b, i++\)\]

In terms of P.R.

\[g = \Pi^1_k\]

\[\text{Sum}(a, 0) = a \quad \text{Sum}(a, m) \rightarrow f(n_1, 0) = n_1 \quad \text{Sum}(a, m+1) = 6(f(n_1, m))\]
Def (version 0):

A. Primitive recursive (PR) function is a function that can be obtained from 0, 1, S, \( \mu \) identity \( \mu, \nu \), \( \mu, \nu \) on \( n \) gives \( 1 \) if we draw a line from this bit on \( n \) and \( \mu \) gives \( 3 \) this is projection.

Pr: \( f(0, \ldots, 0) = 0 \) probably stays.

1. \( 1 = 6(0) \) \( 6(0) \in n + 1 \)
2. \( 2 = 6(6(0)) \) conclusion: to describe which functions are PR.
3. \( 3 = 6(6(6(0))) \) \( \equiv \) composition computable in a loop.

Caution: We are not talking about efficiency. Size grows exponentially.

Unary notation for integers: \( \bar{1} = \bar{1} \), \( \bar{100} = \bar{1} \bar{0} \bar{0} \)

Binary notation: \( \bar{10101} \).

\( \Pi_1^k (n_1, \ldots, n_k) = \Pi_1 \)

Identity in terms of projection: \( \Pi_1 (n) = n \)

Addition: do we need to do basic operation?

\[ a + b = a + \underbrace{1 + \ldots + 1}_b \text{ times} \]

\[ \text{for } (i \leq 0, (i < 0) \Rightarrow (i + 1) + \text{sum}_i) \]

Id = \( \Pi_1 \) (Identity, is projecting a special case).

\( \Pi_1^k (n_1, \ldots, n_k) = \Pi_1 \)

Sum of \( \Pi_1 \) operation is given by:

\[ g = \Pi_1 \]

\[ g(m) = \Pi_1 \]

\[ \text{Sum}(a_0, c) = \underbrace{a_0}_{n_1, n_2} + \text{sum}_{n_1, n_2} \Rightarrow f(n_1, 0) = n_1 \]

\[ \text{Sum}(a_0 + 1) = \text{sum}_{a_0 + 1} \Rightarrow f(n_1, m + 1) = 6(f(n_1, m)) \]
Def. (actual)

A. P.R. function is any function that can be obtained from 0, 6, \( \pi^3 \) by using \( \circ \) and \( \text{FR} \) composition.

result: \( \ast \) is P.R.

Say, \( n,m \in N \)

\[
\begin{align*}
\text{prod} & = 0 ; \\
\text{for}(i=0; i< b; i++) \\
\text{prod} & = \text{prod} + i ; \\
\text{mul}(a, i) & = a ; \\
\text{mul}(a, i+1) & = \text{mul}(a, i) + a .
\end{align*}
\]

\[ f(n, m) = f(n, m + 1) = f(n, m) + m \]

In JAVA terms:

Say, \( n,m \in N \)

\[
\begin{align*}
\text{mul} & = 0 ; \\
\text{for}(i=0; i< m; i++) \\
\text{mul} & = \text{mul} + i ;
\end{align*}
\]

27/11/09

\[
\begin{align*}
\text{mathematical} & \\
\text{term} & \\
\text{mul}(n, m) & = \text{mul}(n, 0) + n \times m \times (n + m) \\
\text{general formula} & \\
\text{mul}(n, m) & = \text{mul}(n, 0) + n \times \sum_{i=1}^{m} i \\
\text{elements} & = \prod_{i=1}^{3} + \prod_{i=1}^{3} \\
\text{mul} & = \text{PR}(0, \text{add}(n^3, \pi^3)) \rightarrow \text{since } f = \text{PR}(g, h) \\
\text{how, add} & = \text{PR}(n^3, 6) \quad \text{here, } g = 0. \\
\text{mul} & = \text{PR}(0, \text{add}(n^3, \pi^3)) \quad \text{projecting term} \\
& = \text{PR}(0, [\text{PR}(n^3, 6)]) \quad \text{projecting term} \quad \text{term}
\end{align*}
\]
\[ n^0 \text{ is PR, } 0^2 = 0 \]

\[ \text{Square}(0) = 0. \]

\[ (m+1)^2 = m^2 + 2m + 1. \quad \text{Square}(m+1) = \text{Square}(m) + m + m + 1 \]

\[ f(0) = 0 \quad \therefore \quad g = 0. \]

\[ f(m+1) = f(m) + m + m + 1 \]

\[ h = f + m + (m+1) \]

\[ = \pi_2^2 + \pi_1^2 + 6 \pi_1^2. \]

\[ h = \text{add} \left( \pi_2^2, \text{add} (\pi_1^2, 6 \pi_1^2) \right) \]

Fact(n) is PR:

\[ \text{fact}(0) = 1 \]

\[ \text{for } (i=0; i<n; i++) \]

\[ \text{fact} = \text{fact} * i; \]

\[ \rightarrow \text{fact}(i++) = (i++) * \text{fact}(i) \]

\[ \text{fact}(0) = 1 \]

\[ \text{fact}(m+1) = \text{fact}(m) * (m+1) \]

\[ f(m, f) = f * (m+1) \]

\[ h = m + f \]

\[ x = \pi_1^2 + \pi_2^2 + \pi_2^2 = \pi_2^2 (\pi_1^2 + 1) \]

\[ \text{fact}(m+1) = \text{fact}(m) * (m+1) \]

\[ \text{fact}(0) = 1 \]

\[ \text{for } (i=1; i<n; i++) \]

\[ \text{fact} = \text{fact} * i; \]

\[ f(m_1, \ldots, m_k, 0) = f(m_1, \ldots, m_k) \]

\[ f(m_1, \ldots, m_k, m+1) = f(m_1, \ldots, m_k) + f(m_1, \ldots, m_k, m) \]

\[ f(0) = 0 \]

\[ f(m+1) = h(m, f(m)) \]

\[ g = 1 \]

\[ h(m, f) = f(m+1). \]
(5) Prove that, \( n^2 + n + 5 \) is P.R.

\[ a - b = \begin{cases} a - b & \text{if } a \geq b \\ 0 & \text{else} \end{cases} \]

\[ \text{Proof: } (a) = 2 \]

Let's prove, \( \text{Prev is P.R.} \)

\[ \text{Prev}(2) = 1 \]
\[ \text{Prev}(0) = 0 \]
\[ \text{Prev}(m+1) = (m+1) \]
\[ m = \pi^2 \]

\[ \text{Prev} = FR(0, \pi^2) \]

Now, \( \text{Prev}(0, \pi^2) \Rightarrow \text{prove this is a function.} \)

In general, \( f(m_1, m_2, m_k, m) = g(m_1, m_2, m_k) \)

\[ f(m_1, m_2, m + 1) = h(m_1, m_2, m, f(m_1, m_2, m)) \]

In the \( \pi^2 \), \( k + 2 = 2 \) (\( h \) is fn of 2 variables).

\[ \pi^2 \]

In the Java prog, \( \text{Prev} = 0; \text{for}(i = 0; i < n; i++) \)

\[ \text{Prev} = i; \]

Prove:

\[ a - b \text{ is P.R.} \]

\[ a - b = (a-1) \times (a^2) \times (a^3) \times \cdots \times (a^{n-1}) \text{ \( \mu \) times} \]

\[ \text{minus}(a, 0) = a \]

\[ \text{minus}(a, m+1) = 1 \text{ \( \mu \) times} \]

\[ \text{minus}(m_1, 0) = m_1 \]

\[ f(m_1, 0) = m_1 \]
\[ \text{add} (m, m+1) = \text{add} (m, m) + 1 \]
\[ n \cdot (m+1) = n \cdot m + m \cdot n \]
\[ n \cdot (m+1) = (m-m) - 1 \]

'0' is PR.

\text{conditions: } \rightarrow \langle, >, =, =\rangle = \text{and, not.}

\text{say}
\begin{align*}
\text{true} & = 1 \\
\text{false} & = 0
\end{align*}
\text{positive}(0) = 0 \text{ (false).}
\text{positive}(m+1) = 1 \text{ if } m > 0 \text{ is P.R.}
\text{PR}(0, 0, 0)

a > b \quad a - b \quad \text{sd be + ve}

a > b \equiv \text{pos}(a - b) \quad \text{PR.}

\text{not is P.R.}

\begin{align*}
\text{not}(o) & = 1 \\
\text{not}(m) & = 1 \text{ or } \text{not}(m) = 1 - m \text{.}
\end{align*}

\begin{align*}
\text{not}(m+1) & = 0 \\
\text{not}(1) & = 0
\end{align*}

\begin{align*}
a 	ext{ \lor } b & \equiv \text{not} \left( \text{not}(a) \text{ \land } \text{not}(b) \right) \text{ (since } y = \text{mul.})
\end{align*}

De Morgan's theory
\begin{align*}
a \lor b & = \neg \left( \neg a \land \neg b \right) \\
\text{hence, } a + b - a \cdot b & = \text{A \lor B.}
\end{align*}

\begin{align*}
\text{not} \left( \text{mul} \left( \text{not}(a), \text{not}(b) \right) \right) \\
\therefore, \quad \forall \text{ is P.R. not, & } \forall \text{ is P.R.}
\end{align*}

\begin{align*}
a = b & = 1 \left( a < b \right) \quad k \ll k \left( b < a \right) \quad \text{PR.}
\end{align*}

\text{\text{true} A \oplus B}
\begin{align*}
0 & 0 \\
0 & 1 \\
1 & 0 \\
1 & 1
\end{align*}
Def: A polynomial is any function that can be obtained from constants & variables, by using +, *

\( (n+1)(n-1) \) is a polynomial.

**Exclusive OR:**

1. \( A \text{ XOR } B \equiv (A \text{ OR } B) \text{ XOR } (A \text{ AND } B) \) [A OR B but not both]

\[= \text{ AND (OR (A,B), not (AND (A,B)))} \]

2. \( A \text{ XOR } B = AB + BA = (A \text{ OR } B) \text{ XOR } (A \text{ AND } B) \)

**If-then statements:**

(a > 0)?: \( A := -A \)

if \((a>0)\) then \(A\) else \(-A\)

Let's assume: \( h(n) = \text{ if } P(n) \text{ then } f(n) \text{ else } g(n) \)

\[P \quad P \quad P \]

We want to prove \( h(n) \) is P.R.

Let's make it natural: Let's do heuristic.

- Either \( P(n) \) and \( f(n) \) is the value
- Or \( \neg P(n) \) and \( g(n) \) is the value.

\[ P(n) \ast f(n) + (\neg P(n) \ast g(n)) \]

\[ P(n) \ast f(n) \text{ and } \neg P(n) \ast g(n) \]

If \( P(n) \) is true \((P=1)\):

- \( f \ast 0 + 0 \ast g = f \)

Else if \( P(n) \) is false \((P=0)\):

- \( 0 \ast f + 1 \ast g = g \)

**Remainder:**

\( 0 \text{ rem } n = 0 \)

\( (m+1) \text{ rem } n = \begin{cases} m \text{ rem } n + 1 & \text{if } m \text{ rem } n + 1 < n \\ 0 & \text{otherwise} \end{cases} \)

\( 0 \text{ rem } 5 = 0 \)
\begin{align*}
\text{rem} (n, m+1) & = \begin{cases} 
\text{rem} (n, m) + 1 < n \text{ then } \text{rem} (n, m) + 1 \mid \quad \text{else } 0 \\
\end{cases} \\
\text{Java form: } & (m \% n + 1 < n) \quad ? \quad (m \% n + 1) : 0 \\
\rightarrow \quad t (m, m+1) & = h (m, m, t (m, m)) \\
& \quad \text{if } (f+1 < m) \text{ then } (f+1) \text{ else } 0. \\
\text{Division} \\
0/n & = 0 \\
(m+1)/n & = \begin{cases} 
(\text{if } (m+1) \% n = 0 \text{ then } (m+1)/n) + 1 \\
\text{else } m/n \\
\end{cases} \\
t (n, 0) & = 0 \\
t (n, m+1) & = \begin{cases} 
(\text{if } (m+1) \% n = 0 \text{ then } (f+1) \text{ else } f) \\
\end{cases} \\
\text{Composition of P.R. functions, then it is P.R.} \\
\text{In Java: } \\
\text{div} = 0; \\
\text{for } i = 0; i < n; i++ \\
\text{if } (i+1) \% n == 0 \\
\text{div}++; \\
\text{Is every computable } f \text{ P.R.?} \\
\text{2 proofs; 1. in more details easier proof, } \\
\text{2. in less details more natural} \\
\text{Theorem: There exists a computable function which is not P.R.} \\
P.R. \text{ in terms of } 0, 0', \pi, P.R. \\
\text{writes: } \\
\text{writing with } \text{LaTeX':} \\
\text{like P.R. } (0, \sigma, \sigma, \sigma, \pi - 1) \rightarrow \text{P.R. } (0, 0, \pi_1) \\
\text{sequence } 0's \& 1's, \text{ how to describe it in natural number?} \\
\text{Example: } \text{sequence } (0, 0, 1) \text{ and } (1, 0, 0).
Strip off front 1's to get the original seq's.

then find out the meaning of seq (which is sigma)

then see whether the seq's are meaningful.

A fn which is computable but not P.R. then again?

\[ f(n) = \begin{cases} \text{if } n \text{ is a valid code} & f_e(n) + 1 \\ \text{else} & 0 \end{cases} \]

---

**Diagram**

- yes
- no

\[ \text{is in a valid code} \]

\[ \text{a P.R. } f_e \]

---

**Statement:** \( f \) is not P.R.

**Proof by contradiction:**

We assume, \( f \) be P.R. then \( f \) is represented by some expression,

\[ \forall n : f(n) = f_e(n) \Rightarrow f(c) = f_e(c) \]

by def: \( f(c) = f_e(c) + 1 \) \( \Rightarrow \)

\[ f(c) = f_e(c) + 1 \]

\[ 0 = 1 \]

not true

---

<table>
<thead>
<tr>
<th>( n )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>P.R. id.</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>P.R. flag</td>
<td>( \times )</td>
<td>( \times )</td>
<td>( \times )</td>
<td>( \times )</td>
<td>( \times )</td>
<td>( \times )</td>
<td>( \times )</td>
<td>( \times )</td>
</tr>
<tr>
<td>( f(1) )</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>( f(4) )</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>( f(2) )</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>( f(3) )</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>( f(0) )</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

---

By applying...
HW: Reproduce the proof \[ \text{computable but not P.R.} \]

**Theorem:** Not every computable \( f \) is P.R.

We will construct a function \( f(n) \) which is \( * \) computable

\( * \) not P.R.

**1st part:** Codes of P.R. \( f \).

Goal is to assign to every P.R. \( f \) a number.

We start with an expression \( \text{PR} \left( \left( \text{PR}^\uparrow 1, \sigma \right) \right) \rightarrow \text{Typeable form} \)

\[
\begin{align*}
11 & = 3 \\
101 & = 5 \\
001 & = 1
\end{align*}
\]

Putting in front:

\[
\begin{align*}
1 & \rightarrow 0111 \\
10 & \rightarrow 10110 \\
11 & \rightarrow 11011
\end{align*}
\]

Thus, computable.

**PR compiler checks that.**

Statement: This is an algorithm that given a natural number \( C \)

* checks whether \( C \) is a valid code of a P.R. \( f \)
* if it is, produces a Java code \( f_c \) for computing the corresponding P.R. \( f \).

**Idea:**

1. Strip off the first "1".
2. Check if all combinations are ASCII symbols.
3. LaTeX compiler checks whether syntax is correct.
4. Check syntax:

\[ \text{PR} \left( \left( \mathcal{P}, \sigma \right) \right) \]

checks for whether \( \text{PR} \left( \left( \mathcal{P}, \sigma \right) \right) \)

then transforms it to \( h = f \).

For \( \mathcal{P} = \{i \in \mathbb{N} : i \text{ is prime} \} \)

\[ h = 4 \]
This shows that $f$ is computable.

\[
\text{Then } f(n) = \frac{1}{n+1} \text{ (see prev. page)}
\]

\[
\text{since } C \text{ is a valid code, by def of } f, \text{ we have:}
\]

\[
\hat{f}(C) = f(C) + 1
\]

\[
\text{in particular, for } x = C, f(C) = f(C)
\]

P.R. but not computable. \quad \hat{f}(n) := \begin{cases} \frac{1}{n+1} & \text{if } n \text{ is a valid code} \\ 0 & \text{otherwise} \end{cases}

Georg Cantor (1845-1918) invented set theory.

Real numbers cannot be enumerated (listed)

All natural nos: \{0, 1, 2, \ldots, n, \ldots\}

All integers: \{0, 1, -1, 2, -2, 3, -3, \ldots\}

All fractions: \{0, \frac{1}{1}, \frac{1}{2}, \frac{2}{1}, \frac{2}{2}, \frac{1}{3}, \frac{2}{3}, \ldots \}

No = listing Real nos. \frac{1}{3} x_0 (0) 1 2 3 4 5 6

\begin{align*}
x_1 : & (1) 1 2 \\
x_2 : & 3 3 3 \\
x_3 : & 1 \quad 4 \quad 1
\end{align*}

Kantor's proof: we only care about the existence.

P.R. proof: we also care about computable diagonalization

\[
f[1, 5, 5, \ldots] \rightarrow \text{This is a construction of Real no which doesn't}
\]

\[
to \text{ the seq.}
\]

Result. \exists a computable $f$ which is not P.R.

1st proof: easier to describe, but $f$ is meaningless.

2nd proof: $f$ is meaningful, but the proof is more difficult.

Reminder: we started with $6$. 
level 0: \( f_0(n, m) = n + 1 \)
level 1: \( f_1(n, m) = f_0(f_1(n, m), m) \)
level 2: \( f_2(n, m) = f_1(f_2(n, m), m) \)
level 3: \( f_3(n, m) = f_2(f_3(n, m), m) \)
level 4: \( f_4(n, m) = f_3(f_4(n, m), m) \)
level 5: \( f_5(n, m) = f_4(f_5(n, m), m) \)
level 6: \( f_6(n, m) = f_5(f_6(n, m), m) \)
level 7: \( f_7(n, m) = f_6(f_7(n, m), m) \)
level 8: \( f_8(n, m) = f_7(f_8(n, m), m) \)
level 9: \( f_9(n, m) = f_8(f_9(n, m), m) \)
level 10: \( f_{10}(n, m) = f_9(f_{10}(n, m), m) \)

\[ f_{k+1}(n, m) = f_k(f_{k+1}(n, m), m) \quad \rightarrow \quad \text{Archimedes} \]

Next times, \( (a^{\text{time}}) = b \)

\[ (2^{a^2}) = 2^a \]

\[ A(n) = f_n(n, n) \]

\[ A(0) = f_0(0, 0) = 1 \]
\[ A(1) = f_1(1, 1) = 2 \]
\[ A(2) = f_2(2, 2) = 4 \]
\[ A(3) = f_3(3, 3) = 3^3 = 27 \]
\[ A(4) = f_4(4, 4) = 4^{16} = 4^{(10^{153})} \]

**Ackermann's Function**

Idea of proving that \( A(n) \) is not P.R.

If \( k \leq k \text{ for loops} \Rightarrow f \leq f_k(n, n) \). 

\( f_k(n) \text{ has no loops} \).

So, with no for loops, \( f_n \text{ constructed} \).

\[ 0, 1, 2, 3, 4, 5, 6, (0), 6(0), 6(6(0)), \ldots \]

\[ n, n+1, n+2, \ldots \]

So, \( \exists C \text{ such that } f(n) \leq n + C \), \( f(m, \ldots, n_k) \leq n_k + C \).
Let's take no variable \( k \),
\[
\begin{align*}
\delta(0) &= q \\
\delta(m+1) &= \max(\delta(m), \beta(m)) + C
\end{align*}
\]
bounded by
\[
\delta(m+1) \leq \max(\delta(m), \beta(m)) + C
\]
Since \( \delta(m+1) \) is not P.R., means there is no for loop
thus, \( \delta(m) \) cannot be a P.R. thus bounded by \( m \)
\[
\begin{align*}
\delta(0) &= q \\
\delta(1) &\leq \max(\delta(0), \beta) + C \leq \delta + C \\
\delta(2) &\leq \max(\delta(1), \beta + C) + C \leq \delta + 2C \\
&\vdots
\end{align*}
\]
\[
\delta(m) \leq \delta + m \cdot C
\]
for no for-loop: \( \delta(m) \leq m + 1 + \cdots + 1 \)
\[
\text{const. times}
\]
for one for-loop
\[
\delta(m) \leq m + \cdots + m + \delta
\]
\[
\text{const. times}
\]

Ackerman \( f_2 \) grows faster than any \( f_k \).
\[
\begin{align*}
\delta(0) &= \frac{n+1}{2} \\
\delta(1) &= n + n = \frac{(2^n)}{2} \\
\delta(2) &= \frac{2^n}{2}
\end{align*}
\]
grows faster than prev. step
And any P.R \( f_k \) is somewhere in btw. this levels.

⇒ Our 1st hypothesis: every computable \( f_k \) is P.R.

Fact: There is one computable \( f_k \) which is not P.R.

Example: Ackermann's \( f_m = A(n) \)

Natural idea: A P.R \( f_k \) is anything that can be obtained from \( 0, \oplus, \odot, \) and \( \odot \) by \( 0 \) and P.R.

Is every computable \( f_k \) A P.R?
what is the list of all computable $f$s

we need to formalize while-loop

while ( \text{true} )
{
    \text{...}
}

$\text{for}$-loop

Big diff: no of iterations is not explicitly known, determined by a condition.

$n$ if iteration $\equiv$ smallest $m$ for which $P(\overline{n}, m)$ is true

mu-recursion

\[ f(n_1, \ldots, n_k) = \mu m. P(n_1, \ldots, n_k, m) \]

smallest $m$ (when the condition fails)

A mu-recursive $f$ is any $f$ which can be obtained from $0, S, \mu$ by $\text{for}$, $\text{PR}$, and $\mu$-recursion.

\[ a \div b = 0 \text{ if } a < b, \quad a - b = c \text{ if } c + b = a \]

$3 \div 5 = 0 \text{ if } a > b$

$5 \div 3 = 2$

$3 \div 5 = \mu c. (c + 5 > 5)$

$2 \div 5 = \mu c. (c + 5 > 2)$

$\mu c. (c + 5 > 2)$

Here we stop at $c = 3$

we stop at $c = 0$

HW: Prove $f$ is mu-recursive $f \equiv \frac{a}{b}$

Infinite loop

\[ f(n) = \mu m. (0 = 1) \text{ if } m = 0 \text{ while } (! (0 = 1)) \text{ This } f \text{ is never defined} \]

We take \( f(n) = 0 \text{ if } m = 0 \)

undefined otherwise.

$\mu m. (n = 0 \text{ if } m = 0)$

if $n = 0 \rightarrow$ return $0$

$n \neq 0 \rightarrow$ undefined.

have $n = 0$ then return $0$. 

\[ \mu m. (n = 0 \text{ if } m = 0) \]
\[ f(n) = \begin{cases} 
1 & \text{if } n = 2 \\
2 & \text{if } n = 3 \\
\text{undefined otherwise} & \end{cases} \]

\[ \mu m \left( (n = 2 \land m = 1) \lor (n = 3 \land m = 2) \right) \]

\[ \text{if } n = 2 : \quad \text{[returns 1]} \]
\[ \text{if } n = 3 : \quad \text{[returns 2]} \]

\[ \mu \text{-recursive fns that may be undefined sometimes} \]

\[ \text{totally recursive} = \mu \text{-recursive and everywhere defined} \]

Every Java program can be described as a \( \mu \)-recursive fn.

Imp. In 1930, 2 diff. definitions appeared:

- recursion (Church) based on a prog. lang.
- Turing: low-level based on step-by-step operations of a computing device.

\[ \text{Turing's \#C} \]

computeable = \( \mu \)-recursive.

Big problem with while-loops:

\[ \text{may not stop (goes into } \infty \text{ loop.)} \]

\[ \text{halt} - \]

Theorem: No algorithm is possible that given a program \( P \) and data \( d \), would check whether \( P \) halts on \( d \) (or not)

we would like to have:

\[ \text{halteq} \rightarrow \text{halteeq} = \begin{cases} 
\text{yes} & \text{if } P \text{ halts on } d \\
\text{no otherwise} & \end{cases} \]
Let's take P, one program.

P in Java (It's in ASCII)

\[ \frac{100}{011} \]

[Java program]

Reduction to contradiction: We assume that halter-checker exists.

We define:

\[
H(n) = \begin{cases} 
\text{ln(n)} + 1 & \text{if } n \text{ is a valid code with Java prog} \\
0 & \text{otherwise} 
\end{cases}
\]

We start with 'n' to test Java compiler decides whether 'n' is syntactically correct.

\(0 \leftarrow \text{check if } m \text{ is a valid code of a Java prog}
\)

\( \text{halterchecker} (H(n, n)) \)

\( \text{run } m \text{ on } n \)

\( \text{add } 1 \)

\( \text{return} \)

It is computable by some Java prog, let's denote by \(c\) the code of this Java prog. \(\forall n (H(n, n) = f(n))\)

By def. of \(f\) since \(c\) is a valid code. In particular for \(x = c\):

\[ f(c) = f(c) \quad \text{(3)} \]

\[ f_c = f_c(c) + 1 \quad \text{(1)} \]

From (0) & (3) Two #s \(f_c(c) \& f_c(c) + 1\) are equal to the same # \(f(c)\), so they must be equal to each other.

\[ f_c(c) = f_c(c) + 1 \]

\[ 0 = 1 \]

Contradicts our assumption.
Diagonalization is relativized.

h.w. 1. P.R. proof if needed.
2. Reproduce the halting proof.
3. N 2.2 proof.
4. Prove that cube checks are impossible.

A program: 1) does it halt? → algo.
                2) if it halts, does it produce the correct result?

we want the \( f_p \), \( \forall x (f(x) = 0) \).

In Java:

```java
public static int zero(int x) {
    return x - x;
}
```

but, \( \frac{\text{return } x}{x - x} \) is not always 0.
when \( x = 0 \), it's undefined.

Def: A zero-checker is a program that, given \( P \) that always halts, checks whether \( \forall x (P(x) = 0) \).

Zero-checker \( (P) = \begin{cases} 
\text{yes} & \text{if } \forall x (P(x) = 0) \\
\text{no otherwise} & 
\end{cases} \)

Theorem: Zero-checkers do not exist.
Proof by contradiction.

We assume that zero-checker exists.

\( \Rightarrow \) we will conclude that a halterchecker exists.

we want:

```
halterchecker: \begin{cases} 
1 & \text{if } P \text{ halts} \text{ and} \\
0 & \text{if } P \text{ doesn't halt} \text{ and} 
\end{cases}
```

we can design:

```
\begin{cases} 
\text{if } P \text{ halts and by time } t \\
\text{otherwise} 
\end{cases} 
```

\( \text{computable} \)?

\( P \) halts on \( d \) \( \equiv \exists t \) \( \text{halt}(P,d,t) \)

\( P \) doesn't halt on \( d \) \( \equiv \forall t / \text{halt}(P,d,t) = 0 \)
If \( \forall t \ (\text{halt}(p,t,d) = 0) \rightarrow p \text{ doesn't halt on } d \),

\( \exists \) otherwise \( p \text{ halts on } d \).

\[ \text{halteffect} \equiv \neg \text{zerocheck} \ (d, p) \]

We know such a function doesn't exist.

Thus \( \exists x \ (p_x) \).

Square-checker \( p \)

\[ \text{Square-checker}(p) = \begin{cases} 1 & \text{if } \forall x \ (p_x(x^2)) \\ 0 & \text{otherwise} \end{cases} \]

Theorem. Square-checker is impossible.

Proof. Assume square-checker exists \( \& \) we will build a zero-checker.

\[ (p_x) \rightarrow \forall x (p_x(x^2)) \]

\[ \begin{array}{c}
\begin{array}{c}
\forall x (p_x(x^2)) \\
\text{square-checker}
\end{array}
\end{array} \]

\[ \begin{array}{c}
\begin{array}{c}
q + x^2 \\
\text{check}
\end{array}
\end{array} \rightarrow \forall x (p_x(x^2)) \]

\[ \begin{array}{c}
\begin{array}{c}
\forall x (p_x + x^2) \\
\text{check}
\end{array}
\end{array} \]

\[ \text{Zero-checker}(p_x) \equiv \text{Square-checker} (p_x + x^2) \]

So, this is a zero-checker.

\[ \forall x (p_x(x^2) = 0) \]

*Note: Functions are not necessarily many-to-one computable.

\[ n \rightarrow \begin{cases} 1 & \text{if } n \text{ is even } n = \frac{n}{2} \\ 1 & \text{if } n \text{ is odd } n = 3n + 1 \end{cases} \]

By an A.P.R, we mean any function that can be obtained from \( \{0, 6, 11\} \) and \( A(n) \) by \( p \) and \( PR \).