Incrementalization:
From Clarity To Efficiency

Y. Annie Liu

Computer Science Department
State University of New York at Stony Brook
At the center of computer science

two major concerns of study:

what to compute

how to compute efficiently

problem solving:

from clear specifications for "what"

to efficient implementations for "how"
From clear specifications to efficient implementations

challenge:

develop a method that is both general and systematic

conflict between clarity and efficiency:

clear specifications usually correspond to straightforward implementations, not at all efficient.

efficient implementations are usually difficult to understand, not at all clear.
iterate: determine a minimum increment to take repeatedly, iteratively, to arrive at the desired program output

incrementalize: make expensive operations incremental in each iteration by using and maintaining useful additional values

implement: design appropriate data structures for efficiently storing and accessing the values maintained

applies to different programming paradigms abstraction
loops: incrementalize none
sets: incrementalize, implement data
recursion: iterate, incrementalize control
rules: iterate, incrementalize, implement both
objects: incrementalize across components module

iterate and incrementalize → integration by differentiation
Loops — a simple example

eliminating multiplications:

\[
i := 1 \quad -- \text{in grid with } a \text{ columns and } b \text{ rows}
\]

\[
\text{while } i \leq b:
\]

\[
... a \times i ... \quad -- \text{access last element of each row}
\]

\[
i := i + 1
\]

strength reduction: an oldest optimization, for array access.
Difference Engine, ENIAC: tabulating polynomials.

need to use language semantics and cost model

exploit algebraic properties: \( a \times (i+1) = a \times i + a \)

store, update, initialize value of \( a \times i \): where? how?
Loops — incrementalize

incrementalize

maintain invariant: $c = a \cdot i$, use and update

\[
i := 1 \quad \rightarrow \quad i := 1; \ c := a;
\]

while $i \leq b$:

\[
\ldots a \cdot i \ldots \quad \rightarrow \quad \ldots c \ldots
\]

\[
i := i + 1 \quad \rightarrow \quad i := i + 1; \ c := c + a;
\]

exploit algebraic properties

maintain additional information

iterate and implement: too little or too much to do
Loops — more examples

**hardware design:** non-restoring binary integer square root

\[
\begin{align*}
  n &:= \text{input}() \\
  m &:= 2^{(l-1)} \\
  \text{for } i &:= 1\text{–}2 \text{ downto } 0: \\
  &\quad p := n - m^2 \\
  &\quad \text{if } p > 0: \\
  &\quad \quad m := m + 2^i \\
  &\quad \text{else if } p < 0: \\
  &\quad \quad m := m - 2^i \\
  \text{output}(m)
\end{align*}
\]

**goal:** a few +- and shifts per bit

**image processing:** blurring

\[
\begin{align*}
\end{align*}
\]

**goal:** a few operations per pixel

**need higher-level abstraction**
Sets — a simple example

graph reachability: edges, source vertices $\rightarrow$ reachable vertices

$$r := s$$

while exists $x$ in $e[r]-r$:

$$r := r \cup \{x\}$$

need to

handle composite set expressions: $x[y]$, $x-y$

design representations of interrelated sets: $e$, $s$, $r$
**Sets — incrementalize and implement**

**incrementalize:** retrieve/add/delete element, test membership

- two invariants for $e[r] \cdot r$: $t = e[r]$, $w = t \cdot r$

**chain rule:** maintain $t$ and then $w$.

- derive rules for maintaining simple and complex invariants.

**implement:** $s$, domain of $e$, range of $e$, $r$, $t$, $w$

**based representations:** records for all elements of related sets;
- a set retrieved from is a linked list of pointers to the records;
- a set tested against is a field in the records.

**iterate:** directly from $\min r: s \subset r$, $r \cup e[r] = r$
query processing: join optimization

\[ r := \{(x,y) : x \in s, y \in t \mid f(x) = g(y)\} \]

iterate:
\[
\begin{align*}
  r &:= \{} \\
  &\text{for } x \text{ in } s: \\
  &\quad r := r \cup \{(x,y) : y \in t \mid f(x) = g(y)\}
\end{align*}
\]

incrementalize: maintain
\[ g_{\text{inverse}} = \{(g(y),y) : y \in t\} \]

derived:
\[
\begin{align*}
  g_{\text{inverse}} &:= \{} \\
  &\text{for } y \text{ in } t: \\
  &\quad g_{\text{inverse}} := g_{\text{inverse}} \cup \{(g(y),y)\} \\
  r &:= \{} \\
  &\text{for } x \text{ in } s: \\
  &\quad \text{for } y \text{ in } g_{\text{inverse}}\{f(x)\}: \\
  &\text{ } r := r \cup \{(x,y)\}
\end{align*}
\]

previous algorithm:
\[
\begin{align*}
  f_{\text{inverse}} &:= \{} \\
  &\text{for } x \text{ in } s: \\
  &\quad f_{\text{inverse}} := f_{\text{inverse}} \cup \{(f(x),x)\} \\
  g_{\text{inverse}} &:= \{} \\
  &\text{for } y \text{ in } t: \\
  &\quad \text{if } g(y) \text{ in domain}(f_{\text{inverse}}): \\
  &\quad \quad g_{\text{inverse}} := g_{\text{inverse}} \cup \{(g(y),y)\} \\
  r &:= \{} \\
  &\text{for } z \text{ in domain}(g_{\text{inverse}}): \\
  &\quad \text{for } x \text{ in } f_{\text{inverse}}\{z\}: \\
  &\text{ } \quad \text{for } y \text{ in } g_{\text{inverse}}\{z\}: \\
  &\text{ } \quad \quad r := r \cup \{(x,y)\}
\end{align*}
\]

compare:
same asymptotic time: \(O(#s + #t + #r)\); fewer loops and ops;
less space: \(O(#t)\) or \(O(\min(#s, #t))\), not \(O(#s + #t)\); simpler and shorter; derived!

role-based access control (RBAC)
core RBAC: 16 expensive queries, 9 kinds, updated in many places.
125 lines python \(\rightarrow\) hundreds of lines. CheckAccess: constant time.
Recursion — a simple example

longest common subsequence: sequences $x$ and $y \rightarrow$ length

\[
lcs(i,j) = \begin{cases} 
0 & \text{if } i=0 \text{ or } j=0 \\
\text{lcs}(i-1,j-1)+1 & \text{if } x[i]=y[j] \\
\text{max}(\text{lcs}(i,j-1),\text{lcs}(i-1,j)) & \text{else} 
\end{cases}
\]

need to

determine how to iterate: recursion to iteration

determine what and how to cache: dynamic programming
Recursion — iterate and incrementalize

\[
lcs(i,j) = \begin{cases} 
0 & \text{if } i=0 \text{ or } j=0 \\
\text{else if } x[i]=y[j] & : \ lcs(i-1,j-1)+1 \\
\text{else} & : \ \max(lcs(i,j-1),lcs(i-1,j)) 
\end{cases}
\]

**iterate:** minimum increment from arguments of recursive calls

\[i,j \rightarrow i+1,j\]

**incrementalize:** cache and use

\[
lcs(i+1,j) \ \text{use } r = lcs(i,j) \rightarrow lcs'(i,j,r) = \begin{cases} 
0 & \text{if } i+1=0 \lor j=0 \\
\text{else if } x[i+1]=y[j] & : \ lcs(i,j-1)+1 \ \text{use } lcs(i,j-1), \text{ cache} \\
\text{else} & : \ \max(lcs(i+1,j-1),lcs(i,j)) \ \text{use } lcs(i,j-1) \\
\end{cases} \rightarrow lcs'(i,j-1,lcs(i,j-1)) \text{ recursively}
\]

**implement:** directly map to recursive or indexed data structures
Recursion — more examples

sequence processing: editing distance, paragraph formatting, matrix chain multiplications, ...

math puzzles: Hanoi tower, find solution in linear time

\[ h(n,a,b,c) \quad -- \text{move } n \text{ disks from } a \text{ to } b \text{ using } c \]
\[ = \text{ if } n \leq 0: \text{skip} \]
\[ \quad \text{else: } h(n-1,a,c,b)::\text{move}(a,b)::h(n-1,c,b,a) \]

iterate: \( n,a,b,c \rightarrow n+1,a,c,b \)
cache: \( \text{hExt}(n,a,b,c) = \langle h(n,a,b,c), h(n,b,c,a), h(n,c,a,b) \rangle \)

\( \text{hExt}(n+1,a,c,b) \) use \( \text{rExt} = \text{hExt}(n,a,b,c) \rightarrow \text{hExt'}(n,a,b,c, \text{rExt}) \)
\[ = \text{ if } n+1 \leq 0: \langle \text{skip},\text{skip},\text{skip} \rangle \]
\[ \text{else: } 1\text{st}(\text{rExt})::\text{move}(a,c)::2\text{nd}(\text{rExt}), \]
\[ 3\text{rd}(\text{rExt})::\text{move}(c,b)::1\text{st}(\text{rExt}), \]
\[ 2\text{nd}(\text{rExt})::\text{move}(b,a)::3\text{rd}(\text{rExt}) \]

simpler than others: maintain 2 additional values, not 5
**Rules — a simple example**

transitive closure:

\[
\text{edge}(u,v) \rightarrow \text{path}(u,v) \\
\text{edge}(u,w), \text{path}(w,v) \rightarrow \text{path}(u,v)
\]

need to

find a way to proceed
determine what and how to maintain
design representations of different kinds of facts

additional question

can we give time and space complexity guarantees?
**Rules — iterate, incrementalize, implement**

**iterate**: add one fact at a time until fixed point is reached

**incrementalize**: maintain maps indexed by shared arguments

**implement**: design nested linked lists and arrays of records

**time and space guarantees**:

\[
\begin{align*}
\text{edge}(u,v) & \rightarrow \text{path}(u,v) \\
\text{edge}(u,w), \text{path}(w,v) & \rightarrow \text{path}(u,v)
\end{align*}
\]

**time**: \# of combinations of hypotheses — optimal

\[
O(\min(\#\text{edge} \times \#\text{path}.2/1, \#\text{path} \times \#\text{edge}.1/2))
\]

**space**: \(O(\#\text{edge})\), for storing inverse map of edge
program analysis: dependence analysis, pointer analysis, information flow analysis, ...

trust management: SPKI/SDSI authorization

\[\text{auth}(k_1, [k_2], \text{TRUE}, a_1, v_1), \text{auth}(k_2, s_2, d_2, a_2, v_2) \rightarrow \text{auth}(k_1, s_2, d_2, \text{PInt}(a_1, a_2), \text{VInt}(v_1, v_2))\]

\[\text{auth}(k_1, [k_2 \ [n_2 \ ns_3]], d, a, v_1), \text{name}(k_2, n_2, [k_3], v_2) \rightarrow \text{auth}(k_1, [k_3 \ ns_3], d, a, \text{VInt}(v_1, v_2))\]

\[\text{name}(k_1, n_1, [k_2 \ [n_2 \ ns_3]], v_1), \text{name}(k_2, n_2, [k_3], v_2) \rightarrow \text{name}(k_1, n_1, [k_3 \ ns_3], \text{VInt}(v_1, v_2))\]

find authorized keys: \(O(in * kp * kn)\), better than \(O(in * k * k)\).
Objects — a simple example

the “what” of a software component:

queries: compute information using data w/o changing data.

updates: change data.

example:

class LinkedList in Java has many methods:

size(), 18 add or remove, several other queries.
how to implement the queries and updates: \textit{varies significantly}

\textbf{straightforward}: \\
queries compute requested information. \\
updates change base data. \\
\textbf{example}: \texttt{size()} contains a loop that computes the size.

\textbf{observe}: \\
queries are often repeated, many are easily \textit{expensive}; \\
updates can be frequent, they are usually \textit{small}.

\textbf{sophisticated — incrementalized}: \\
store derived information; queries return stored information. \\
updates also update stored information. \\
\textbf{example}: maintain size in a field, and update it in 18 places.
Objects — more examples

examples: wireless protocols, electronic health records, virtual reality, games, ...

findStrongSignals(): return \{s in signals | s.getStrength() > threshold\}

class Protocol
    signals: set(Signal)
    threshold: float
    + strongSignals: set(Signal)
    ...
    addSignal(signal): signals.add(signal)
    + signal.takeProtocol(this)
    + if signal.getStrength() > threshold
    + strongSignals.add(signal)
    * findStrongSignals(): return strongSignals
    + updateSignal(signal):
    + if signals.contains(signal)
    + if strongSignals.contains(signal)
    + if not signal.getStrength() > threshold
    + strongSignals.remove(signal)
    + else
    + if signal.getStrength() > threshold
    + strongSignals.add(signal)
...

findStrongSignal: \(O(\#\text{signals}) \rightarrow O(1)\). setStrength: \(O(1) \rightarrow O(\#\text{protocols})\).
Iterate, Incrementalize, Implement

iterate at a minimum increment step; incrementalize expensive computations; implement on efficient data structures.

loops iter, inc, impl
maintaining invariants, algebraic properties, additional values

sets iter, inc, impl
chain rule, deriving maintenance rules; based representations

recursion iter, inc, impl
recursion to iteration; dynamic programming

rules iter, inc, impl
all, giving time and space complexity guarantees

objects iter, inc, impl
all, across components

connect theory w/ practice. like differentiation & integration.
References

loops  [Allen69..., Liu97, LS98a/LSLR05...]

sets  [Earley76..., PK82, Willard96, Willard02, LWGRCZZ06...]

recursion  [BD77..., Smith90, LS99/03, LS00, LS02/09...]

rules  [Forgy82, Vardi82..., McAllester99, LS03/09...]

objects  [..., LSGRL05, RL08...]

more:  Systematic Program Design: From Clarity to Efficiency
Ongoing projects

objects and nested sets: generating incremental implementations of queries with complexity guarantees

datalog and extensions: generating efficient implementations for demand-driven queries [PPDP 10, SIGMOD 11], functors, and arbitrary negation

logic quantifications: generating optimized implementations by reducing nesting levels and using aggregates [OOPSLA 12]

distributed algorithms: from very high-level specifications to efficient implementations [OOPSLA 12, SSS 12]