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.

A general and systematic method

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 $\rightarrow$ integration by differentiation

Loops — a simple example

eliminating multiplications:

```
i:=1                   -- in grid with a columns and b rows
while i <= b:
  :
  ...a*i...            -- 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*(i+1)` = `a*i+a`

store, update, initialize value of `a*i`: where? how?

Loops — incrementalize

incrementalize

maintain invariant: `c` $=$ `a*i`, use and update

```
i:=1               → i:=1; c:=a;
while i <= b:
  :
  ...a*i...        → ...c...
  :
  i:=i+1           → 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

```
n := input()
m := 2^(l-1)
for i := l-2 downto 0:
  p := n - m^2
  if p > 0:
    m := m + 2^i
  else if p < 0:
    m := m - 2^i
output(m)
```

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

image processing: blurring

```
 _______________
|   ______      |
| |       |     |
| |       |     |
| |       |     |
| |______|      |
|               |
|_______________|
```

`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 U {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]-r`: `t` = `e[r]`, `w` = `t-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 U e[r] = r`

Sets — more examples

query processing: join optimization

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

iterate:

```
r := {}
for x in s:
  r := r U {[x,y]: y in t | f(x)=g(y)}
```

incrementalize: maintain

```
ginverse = {[g(y),y]: y in t}
```

derived:

```
ginverse := {}
for y in t:
  ginverse = ginverse U {[g(y),y]}
r := {}
for x in s:
  for y in ginverse{f(x)}
    r := r U {[x,y]}
```

previous algorithm:

```
finverse := {}
for x in s:
  finverse := finverse U {[f(x),x]}
ginverse := {}
for y in t:
  if g(y) in domain(finverse):
    ginverse := ginverse U {[g(y),y]}
r := {}
for z in domain(ginverse):
  for x in finverse{z}:
    for y in ginverse{z}:
      r := r U {[x,y]}
```

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)
= if i=0 or j=0: 0
  else if x[i]=y[j]: lcs(i-1,j-1)+1
  else: max(lcs(i,j-1),lcs(i-1,j))
```

need to

determine how to iterate: recursion to iteration

determine what and how to cache: dynamic programming

Recursion — iterate and incrementalize

```
lcs(i,j)
= if i=0 or j=0: 0
  else if x[i]=y[j]: lcs(i-1,j-1)+1
  else: max(lcs(i,j-1),lcs(i-1,j))
```

iterate: minimum increment from arguments of recursive calls

`i,j` → `i+1,j`

incrementalize: cache and use

```
lcs(i+1,j)   use r = lcs(i,j)          → lcs'(i,j,r)
= if i+1=0 lor j=0: 0
  else if x[i+1]=y[j]: lcs(i,j-1)+1  use lcs(i,j-1), cache
  else: max(lcs(i+1,j-1),lcs(i,j))   use lcs(i,j-1)
                                     → lcs'(i,j-1,lcs(i,j-1))
                                     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)              -- move n disks from a to b using c
= if n<=0: skip
  else: h(n-1,a,c,b)::move(a,b)::h(n-1,c,b,a)
```

iterate: `n,a,b,c` $\rightarrow$ `n+1,a,c,b`
cache: `hExt(n,a,b,c) = <h(n,a,b,c), h(n,b,c,a), h(n,c,a,b)>`

```
hExt(n+1,a,c,b)   use rExt=hExt(n,a,b,c)  → hExt'(n,a,b,c,
= if n+1<=0: <skip,skip,skip>                         rExt)
  else: 1st(rExt)::move(a,c)::2nd(rExt),
        3rd(rExt)::move(c,b)::1st(rExt),
        2nd(rExt)::move(b,a)::3rd(rExt)>
```

simpler than others: maintain 2 additional values, not 5

Rules — a simple example

transitive closure:

```
edge(u,v) -> path(u,v)
edge(u,w), path(w,v) -> 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:

```
edge(u,v) -> path(u,v)
edge(u,w), path(w,v) -> path(u,v)
```

time: # of combinations of hypotheses — optimal

$O(\min(\#\texttt{edge} \times \#\texttt{path.2/1},\ \#\texttt{path} \times \#\texttt{edge.1/2}))$

edges vertices output indegree

space: $O(\#\texttt{edge})$, for storing inverse map of `edge`

Rules — more examples

program analysis: dependence analysis, pointer analysis, information flow analysis, ...

trust management: SPKI/SDSI authorization

```
auth(k1,[k2],TRUE,a1,v1), auth(k2,s2,d2,a2,v2)
        -> auth(k1,s2,d2,PInt(a1,a2),VInt(v1,v2))

auth(k1,[k2 [n2 ns3]],d,a,v1), name(k2,n2,[k3],v2)
        -> auth(k1,[k3 ns3],d,a,VInt(v1,v2))

name(k1,n1,[k2 [n2 ns3]],v1), name(k2,n2,[k3],v2)
        -> name(k1,n1,[k3 ns3],VInt(v1,v2))
```

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.

Objects — incrementalize

how to implement the queries and updates: varies significantly

straightforward:

queries compute requested information.

updates change base data.

example: `size()` contains a loop that computes the size.

observe:

queries are often repeated, many are easily expensive;

updates can be frequent, they are usually small.

sophisticated — incrementalized:

store derived information; queries return stored information.

updates also update stored information.

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
+           strongSingals.remove(signal)
+       else
+         if signal.getStrength()>threshold
+           strongSingals.add(signal)
    ...
```

```
  class Signal
    strength: float
+   protocols: set(Protocol)
    ...
+   takeProtocol(protocol):
+     protocols.add(protocol)
    setStrength(v):
      strength = v
+     for protocol in protocols
+       protocol.updateSignal(this)
    getStrength(): return strength
    ...
  ...
```

	original lines
*	changed lines
+	added lines

`findStrongSignal`: O(#signals) $\rightarrow$ O(1). `setStrength`: O(1) $\rightarrow$ O(#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
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]