Iterate, incrementalize, and implement: A systematic approach to efficiency improvement and guarantees

Y. Annie Liu

Computer Science Department
State University of New York at Stony Brook

Incremental Programs

f' is an incremental version of f under $\oplus$
if f' computes $f(x \oplus y)$ efficiently by making use of $f(x)$.

examples:

- f is $sort$, x is a list of numbers, $x \oplus i$ is $cons(i, x)$.
  if $sort(x) = r$, then $sort'(x, i, r) = sort(cons(i, x))$.
- f is C compiler, x is C program, $\oplus$ is change to program.
- f is loop body, x is ind. variable, $\oplus$ is increment to ind. var.

Incrementalization:

Essential for Efficiency Improvement

in essence,

all non-trivial computation proceeds in an repetitive fashion.

for efficiency,

each iteration needs to be computed incrementally using the stored results of the previous iteration.

incrementalization: iteration body $\longrightarrow f$

iteration increment $\longrightarrow \oplus$

an incremental version can be used to replace the iteration body.

A **General Systematic** Transformational Approach

given f and $\oplus$,

derive an incremental program
that computes $f(x \oplus y)$ efficiently by using

P1. the value of $f(x)$,
P2. the intermediate results of $f(x)$, and
P3. auxiliary information of $f(x)$ that can be inexpensively maintained

greater incrementality

Work in Incremental Computation

incremental algorithms (dynamic, on-line)

parsing, AG evaluation, data-flow analysis, circuit evaluation, constraint solving, transitive closure, shortest path, MST, connectivity, scheduling...

ad hoc

incremental execution frameworks

incremental AG evaluation framework, function caching, lambda reduction, traditional partial evaluation, INC, program abstraction...

poor specializability

incremental-program derivation approaches

many methods for program efficiency improvement in optimizing compilers, transformational programming, and programming methodology

limited primitive operators or only high-level strategies

Outline

- incrementalization: P1: exploiting previous result
  P2: caching intermediate results
  P3: discovering auxiliary information
- more examples
- optimization: iterate, incrementalize, implement
  Loops: optimize loops & aggregate array computations
  Recursion: opt rec funs & rec data structs & to iteration
  Sets: implement sets & fixed points
  Rules: implement relations & rules
- summary, and prototype system CACHET

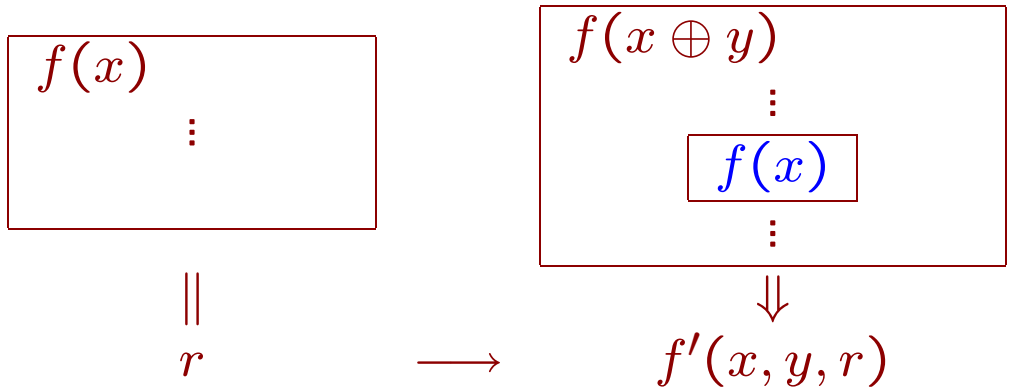
Language and Example

a simple language: functional, first-order, call-by-value.

$$
\begin{array}{lcl}
cmp(x) & = & sum(odd(x)) \leq prod(even(x)) \\
\\
odd(x) & = & \textbf{if } null(x) \textbf{ then } nil \\
 & & \textbf{else } cons(car(x), even(cdr(x))) \\
even(x) & = & \textbf{if } null(x) \textbf{ then } nil \\
 & & \textbf{else } odd(cdr(x)) \\
sum(x) & = & \textbf{if } null(x) \textbf{ then } 0 \\
 & & \textbf{else } car(x) + sum(cdr(x)) \\
prod(x) & = & \textbf{if } null(x) \textbf{ then } 1 \\
 & & \textbf{else } car(x) * prod(cdr(x)) \\
\\
x \oplus y & = & cons(y, x)
\end{array}
$$

cost model: asymptotic time. goal: 1. reduce time, 2. save space.

P1. Exploiting Previous Result

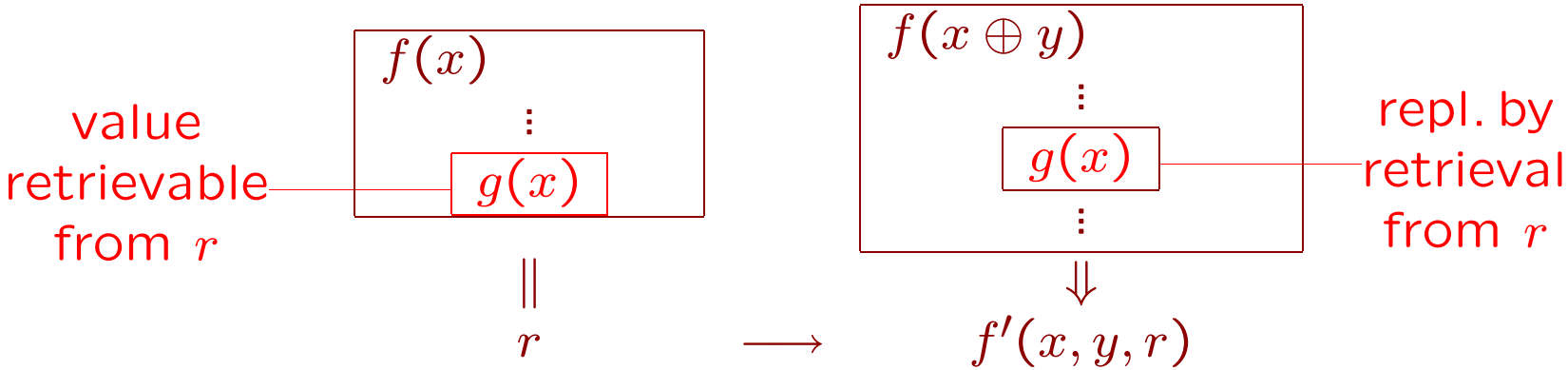


$$f(x) = r \qquad f'(x, y, r) = f(x \oplus y)$$

- introduce function with cached result as extra argument
- unfold (expand) and simplify
- replace using cached result
  — use identity
- replace function call using introduced incremental version

[LT95]: *Science of Computer Programming* 24 (1) 1–39, February 1995.

P1. Exploiting Previous Result



$$f(x) = r \qquad f'(x, y, r) = f(x \oplus y)$$

- introduce function with cached result as extra argument
- unfold (expand) and simplify
- replace using cached result

  — use equality reasoning & auxiliary specialization
- replace function call using introduced incremental version

[LT95]: *Science of Computer Programming* 24 (1) 1–39, February 1995.

P1. Exploiting Previous Result—Example

$$\boxed{\begin{array}{ll} sum(x) = & \textbf{if } null(x) \textbf{ then } 0 \\ & \textbf{else } car(x) + sum(cdr(x)) \\ \\ x \oplus y = & cons(y, x) \end{array}}$$

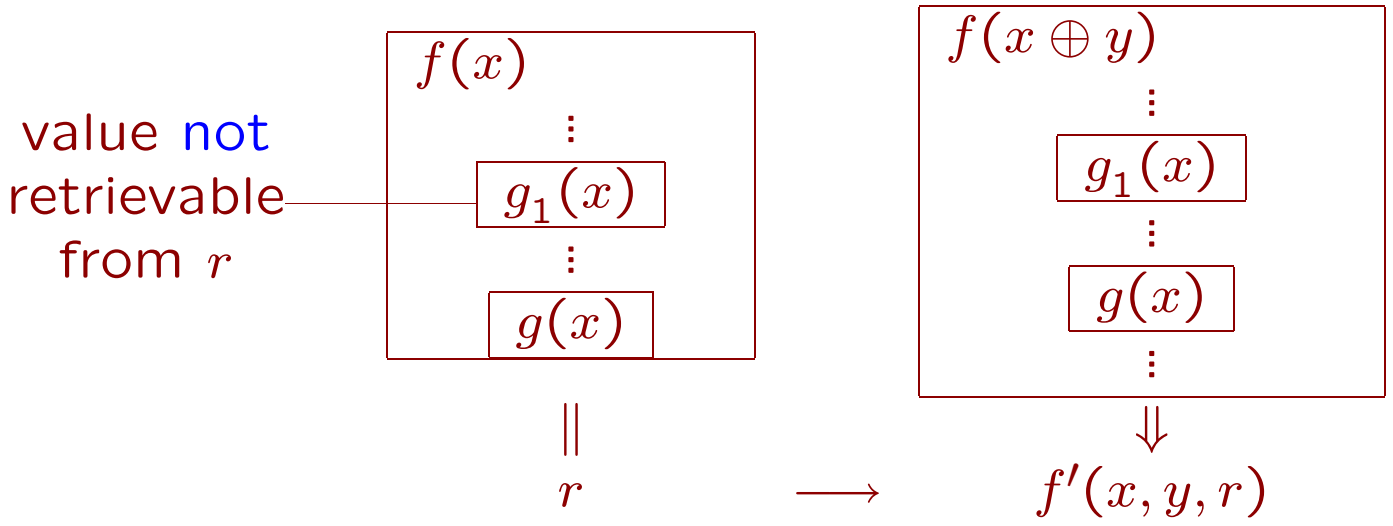
if $sum(x) = r$, then $sum'(x, y, r) = sum(cons(y, x))$.

$$\begin{array}{l} \textbf{if } null(cons(y, x)) \textbf{ then } 0 \\ \textbf{else } \underbrace{car(cons(y, x))}_{y} + sum(\underbrace{cdr(cons(y, x))}_{x}) \end{array}$$

$$\boxed{sum'(\not{x}, y, r) = y + r}$$

$O(n) \longrightarrow O(1)$

P2. Caching Intermediate Results



$$f(x) = r \qquad f'(x, y, r) = f(x \oplus y)$$
$$\hat{f}(x) = \hat{r} \qquad \hat{f}'(x, y, \hat{r}) = \hat{f}(x \oplus y)$$

3 stages: cache & prune

I: cache all intermediate results of $f \Rightarrow \bar{f}$

II: incrementalize $\bar{f}$ under $\oplus$ using P1 $\Rightarrow \bar{f}'$

III: prune $\bar{f}$, $\bar{f}'$ using dependency in $\bar{f}' \Rightarrow \hat{f}$, $\hat{f}'$

[LT95]: *ACM SIGPLAN PEPM.* [LST98]: *ACM Trans. Program. Lang. Syst.*

P2. Caching Intermediate Results—Example

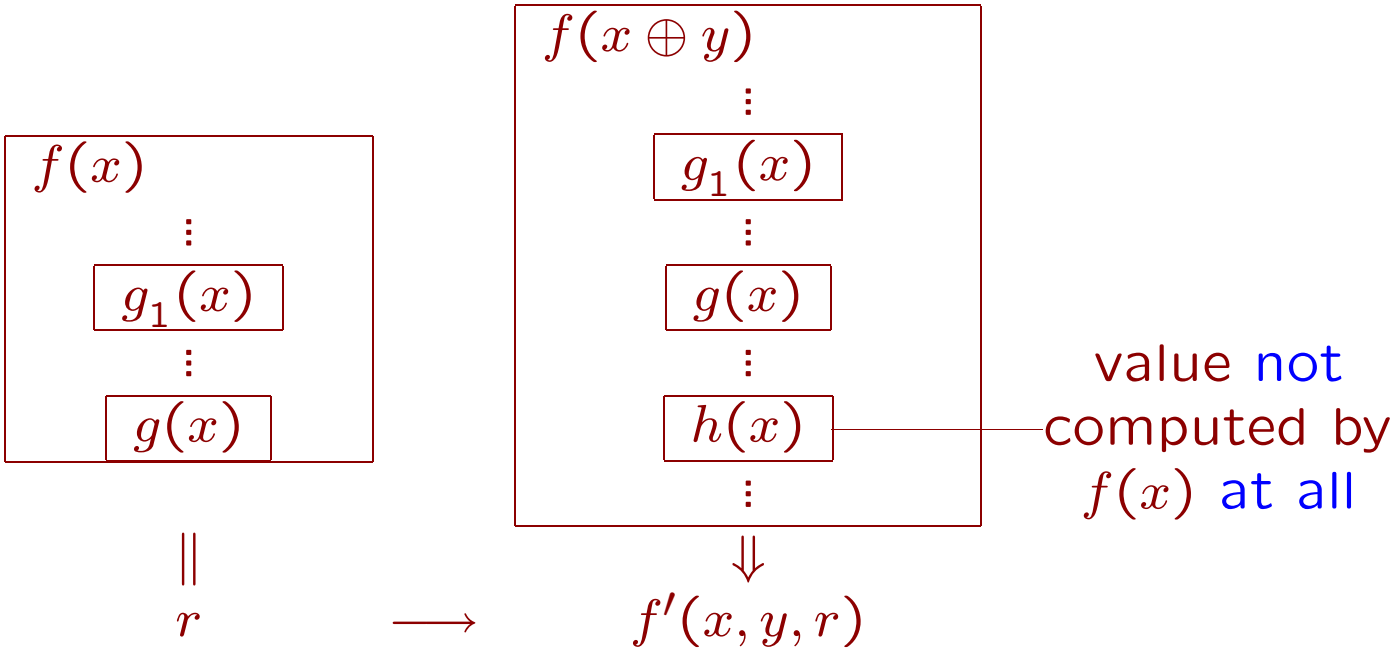
$$\begin{array}{l} cmp(x) = \ sum(odd(x)) \leq prod(even(x)) \\ \\ x \oplus \langle y_1, y_2 \rangle = \ cons(y_1, cons(y_2, x)) \end{array}$$

if $\widehat{cmp}(x) = \hat{r}$, then $\widehat{cmp}'(y_1, y_2, \hat{r}) = \widehat{cmp}(cons(y_1, cons(y_2, x)))$.

$$\begin{array}{l} \widehat{cmp}(x) = \ \textbf{let } v_1 = sum(odd(x)) \textbf{ in} \\ \qquad\qquad\quad \textbf{let } v_2 = prod(even(x)) \textbf{ in} \\ \qquad\qquad\quad < v_1 \leq v_2, \ v_1, \ v_2 > \\ \\ \widehat{cmp}'(y_1, y_2, \hat{r}) = \ \textbf{let } v_1 = y_1 + 2nd(\hat{r}) \textbf{ in} \\ \qquad\qquad\qquad\quad \textbf{let } v_2 = y_2 * 3rd(\hat{r}) \textbf{ in} \\ \qquad\qquad\qquad\quad < v_1 \leq v_2, \ v_1, \ v_2 > \end{array}$$

$O(n) \longrightarrow O(1)$

P3. Discovering Auxiliary Information



$$f(x) = r \qquad f'(x, y, r) = f(x \oplus y)$$
$$\tilde{f}(x) = \tilde{r} \qquad \tilde{f}'(x, y, \tilde{r}) = \tilde{f}(x \oplus y)$$

2 phases:

A: discover candidate auxiliary information using P1

B: use candidate auxiliary information using P2

[LST96]: *ACM SIGPLAN POPL*. [LST01]: *Science of Computer Programming*.

P3. Discovering Auxiliary Information—Example

$$cmp(x) = \; sum(odd(x)) \leq prod(even(x))$$

$$x \oplus y = \; cons(y, x)$$

if $\widetilde{cmp}(x) = \tilde{r}$, then $\widetilde{cmp}'(y, \tilde{r}) = \widetilde{cmp}(cons(y, x))$.

$$\begin{array}{l} \widetilde{cmp}(x) = \; \textbf{let } v_1 = odd(x) \textbf{ in} \\ \qquad\qquad\quad \textbf{let } u_1 = sum(v_1) \textbf{ in} \\ \qquad\qquad\quad \textbf{let } v_2 = even(x) \textbf{ in} \\ \qquad\qquad\quad \textbf{let } u_2 = prod(v_2) \textbf{ in} \\ \qquad\qquad\quad < u_1 \leq u_2,\, u_1,\, u_2,\, sum(v_2),\, prod(v_1) > \\ \\ \widetilde{cmp}'(y, \tilde{r}) = \quad < y{+}4th(\tilde{r}) \leq 5th(\tilde{r}), \\ \qquad\qquad\qquad\quad y{+}4th(\tilde{r}),\, 5th(\tilde{r}),\, 2nd(\tilde{r}),\, y * 3rd(\tilde{r}) > \end{array}$$

$O(n) \longrightarrow O(1)$

More Examples

problem	batch	inc
insertion sort	$O(n^2)$	$O(n)$
selection sort	$O(n^2)$	$O(n)$
merge sort	$O(n \log n)$	$O(n)$

problem	naive	inc-ed
Fibonacci function	$O(2^n)$	$O(n)$

Example Derivation: Insertion Sort $\longrightarrow$ Insertion

$$
\begin{array}{lcl}
sort(x) & = & \textbf{if } null(x) \textbf{ then } nil \\
 & & \textbf{else } insert(car(x), sort(cdr(x))) \\
 & & \\
insert(i,x) & = & \textbf{if } null(x) \textbf{ then } cons(i, nil) \\
 & & \textbf{else if } i \leq car(x) \textbf{ then } cons(i,x) \\
 & & \textbf{else } cons(car(x), insert(i, cdr(x))) \\
 & & \\
x \oplus y & = & cons(y,x)
\end{array}
$$

if $sort(x) = r$, then $sort'(x,y,r) = sort(cons(y,x))$.

$$
\begin{array}{l}
\textbf{if } null(cons(y,x)) \textbf{ then } nil \\
\textbf{else } insert(\underbrace{car(cons(y,x))}_{y}, sort(\underbrace{cdr(cons(y,x))}_{x}))
\end{array}
$$

$$
\boxed{sort'(\not{x}, y, r) = insert(y, r)}
$$

$O(n^2) \longrightarrow O(n)$

Example Derivation: Selection Sort → Insertion

```
FUNCTION DEFINITIONS:
sort(x) =
  if null(x) then
    nil
  else
    let k = least(x)
    in
      cons(k,sort(rest(x,k)))
    end;

FUNCTION TO BE EVALUATED:
sort

OLD INPUT TO THE FUNCTION:
x

NEW INPUT TO THE FUNCTION:
cons(i,x)
```

```
least(x) =
  if null(cdr(x)) then
    car(x)
  else
    let s = least(cdr(x))
    in
      if car(x) < s then
        car(x)
      else
        s
    end;

rest(x, k) =
  if k == car(x) then
    cdr(x)
  else
    cons(car(x),rest(cdr(x),k));
```

sort'(i,x,r) = sort(cons(i,x)) for r = sort(x)

sort(cons(i,x)) — expand & simplify

definition

```
= if null(cons(i, x)) then
    nil
  else
    let k = least(cons(i,x))
    in
      cons(k,sort(rest(cons(i,x),k)))
    end
```

simplify

```
= let k = least(cons(i, x))
  in
    cons(k,sort(rest(cons(i,x),k)))
  end
```

expand & simplify (cont'd)

least(cons(i,x))

definition

```
= if null(cdr(cons(i,x))) then
    car(cons(i, x))
  else
    let s = least(cdr(cons(i,x)))
    in
      if car(cons(i,x)) < s then
        car(cons(i,x))
      else
        s
    end
```

simplify

```
= if null(x) then
    i
  else
    let s = least(x)
    in
      if i < s then
        i
      else
        s
    end
```

sort(cons(i,x)) cont'd (I)

expand & simplify (cont'd)

expand

```
= let k = if null(x) then
            i
          else
            let s = least(x)
            in
              if i < s then
                i
              else
                s
              end
  in
    cons(k,sort(rest(cons(i,x),k)))
  end
```

simplify

```
= if null(x) then
    cons(i,sort(rest(cons(i,x),i)))
  else
    let s = least(x)
    in
      if i < s then
        cons(i,sort(rest(cons(i,x),i)))
      else
        cons(s,sort(rest(cons(i,x),s)))
      end
```

expand & simplify (cont'd)

rest(cons(i,x),i)

definition

```
= if i == car(cons(i,x)) then
    cdr(cons(i,x))
  else
    cons(car(cons(i,x)),
         rest(cdr(cons(i,x)),i))))
```

simplify

```
= if i == i then
    x
  else
    cons(i,rest(x,i))
```

simplify

```
= x
```

rest(cons(i,x),s)

definition

```
= if s == car(cons(i,x)) then
    cdr(cons(i,x))
  else
    cons(car(cons(i,x)),
         rest(cdr(cons(i,x)),s))))
```

simplify

```
= if s == i then
    x
  else
    cons(i,rest(x,s))
```

sort(cons(i,x)) cont'd (II)

expand & simplify (cont'd)

unfold

```
= if null(x) then
    cons(i,sort(x))
  else
    let s = least(x)
    in
      if i < s then
        cons(i,sort(x))
      else
        cons(s,sort(if s == i then
                      x
                    else
                      cons(i,rest(x,s))))
    end
```

simplify

```
= if null(x) then
    cons(i,sort(x))
  else
    let s = least(x)
    in
      if i <= s then
        cons(i,sort(x))
      else
        cons(s,sort(
                 cons(i,rest(x,s))))
    end
```

sort'(i,x,r) = sort(cons(i,x)) for r = sort(x)

```
= if null(x) then
    cons(i,sort(x))
  else
    let s = least(x)
    in
      if i <= s then
        cons(i,sort(x))
      else
        cons(s,sort(cons(i,rest(x,s))))
    end
```

```
sort(x) =
  if null(x) then
    nil
  else
    let s = least(x)
    in
      cons(s,sort(rest(x,s)))
    end;
```

sort(x) = r, null(x) = null(r)

if null(x) = false, then
least(x) = s = car(r),
sort(rest(x,s)) = cdr(r)

equality reasoning &
auxiliary specialization

sort(cons(i,rest(x,s))) = sort'(i,rest(x,s),cdr(r))

sort'(i,x,r) = sort(cons(i,x)) for r = sort(x)

replace using cached result

```
sort'(i,x,r) =
  if null(r) then
    cons(i,r)
  else
    let s = car(r)
    in
      if i <= s then
        cons(i,r)
      else
        cons(s,sort'(i,rest(x,s),cdr(r)))
    end
```

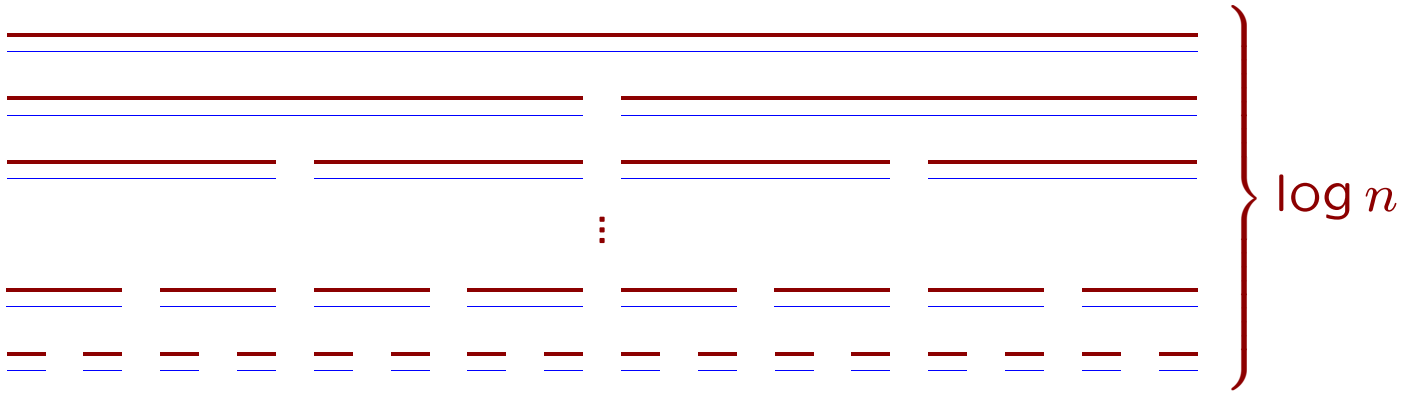
eliminate dead code

```
sort'(i,r) =
  if null(r) then
    cons(i,r)
  else
    let s = car(r)
    in
      if i <= s then
        cons(i,r)
      else
        cons(s,sort'(i,cdr(r)))
    end
```

Example: Merge Sort $\longrightarrow$ Incremental Merge Sort

$$sort(x) = \textbf{if } null(x) \textbf{ then } nil \textbf{ else if } null(cdr(x)) \textbf{ then } x \textbf{ else } merge(sort(odd(x)), sort(even(x)))$$

$$sort'(y, r) = merge(cons(y, nil), r)$$

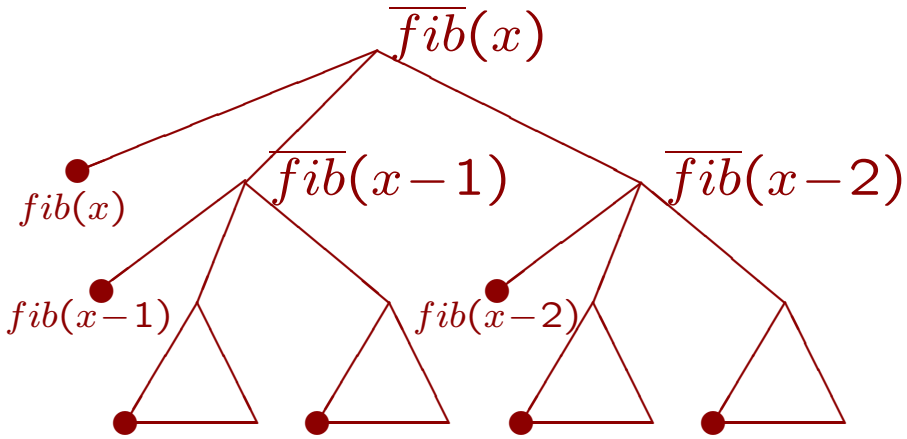


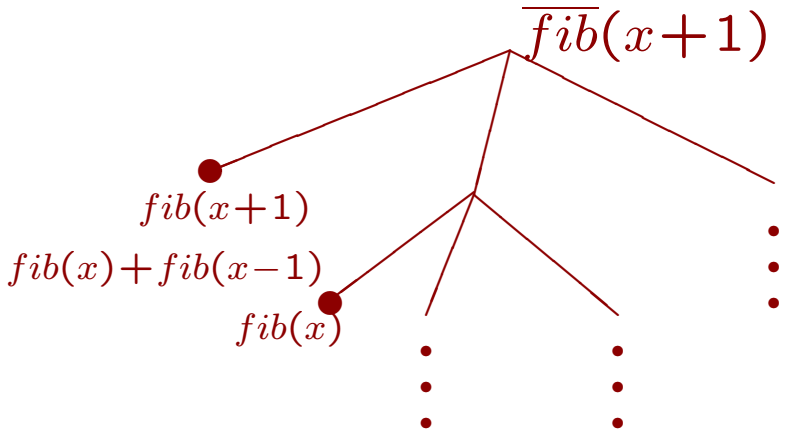
$$\widehat{sort}'(y, \hat{r}) = \textbf{if } null(1st(\hat{r})) \textbf{ then } <cons(y, nil)> \textbf{ else if } null(cdr(1st(\hat{r}))) \textbf{ then } <merge(cons(y,nil), 1st(\hat{r})), <cons(y,nil)>, <1st(\hat{r})>> \textbf{ else let } u_1 = \widehat{sort}'(y, 3rd(\hat{r})) \textbf{ in let } u_2 = 2nd(\hat{r}) \textbf{ in } <merge(1st(u_1), 1st(u_2)), u_1, u_2>$$

$O(n \log n) \longrightarrow O(n)$

Example: Exponential Fibonacci → Linear Fibonacci

$$fib(x) = \textbf{if } x \leq 1 \textbf{ then } 1 \textbf{ else } fib(x-1) + fib(x-2)$$





$$fib_1(x) = \textbf{if } x \leq 1 \textbf{ then } <1,1> \textbf{ else let } r_1 = fib_1(x-1) \textbf{ in } <1st(r_1)+2nd(r_1), 1st(r_1)>$$

$O(2^n) \longrightarrow O(n)$

Iterate, Incrementalize, and Implement

Loops — iter, inc, impl

optimize loops;

optimize aggregate array computations in loops

Recursion — iter, inc, impl

optimize recursive functions & recursive data structures;

transform recursion to iteration

Sets — iter, inc, impl

implement sets, maps, & fixed points

Rules — iter, inc, impl

implement relations & rules

Loops. Optimize Loops

- expensive computations in loop body $\longrightarrow$ f
  updates to parameters of such computations $\longrightarrow$ $\oplus$
- exploit algebraic properties of primitives
- example: non-restoring binary integer sqrt, for VLSI design

$n := input;\ m := 2^{l-1};$
for $i := l-2$ **downto** 0 **do**
  $p := n - m^2;$
  if $p > 0$ **then**
    $m := m + 2^i$
  else if $p < 0$ **then**
    $m := m - 2^i;$
$output(m)$

$m^2, 2^i \longrightarrow +, -, *2, /2$

manually derived, ad hoc;
proved w/Nuprl [OLHA95]

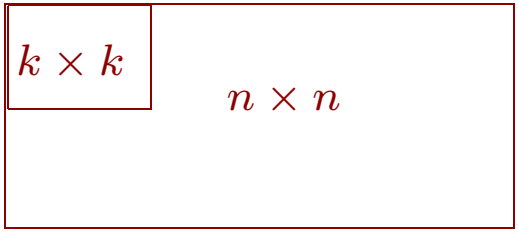
vs.

systematic and automated;
found unnecessary shift and
opt's delayed to hardware

[Liu97]: *IFIP TC2.* [JLZ99]: *CHARME.* [JLZ03]: *Intl. J. Soft. Tools Tech. Transfer.*

Loops. Optimize Aggregate Array Comps

- aggregate array computations $\longrightarrow$ f
  updates to array subscripts by loops $\longrightarrow$ $\oplus$
- reduce to constraint simplification; use Omega
- example: local summation problem, in image processing




```
for i := 0 to n−k do
  for j := 0 to n−k do
    s[i,j] := 0;
    for k := 0 to k − 1 do
      for l := 0 to k − 1 do
        s[i,j] := s[i,j] + a[i+k,j+l]
```

a small portion

```
...
for i := 1 to n−k do
  for j := 1 to n−k do
    b[i−1+k,j] := b[i−1+k,j−1]
                  −a[i−1+k,j−1]
                  +a[i−1+k,j−1+k];
    s[i,j] := s[i−1,j]
              −b[i−1,j]
              +b[i−1+k,j]
...
```

[LS98]: *IEEE Intl. Conf. Computer Langs.* [LSLR03]: *TOPLAS minor revision.*

Recursion. Optimize Recursive Functions

- general (non-linear) recursive functions $\longrightarrow$ f
  minimal increments in arguments of recursive calls $\longrightarrow$ $\oplus$
- achieve dynamic programming straightforwardly
- examples, with improvements in running times

binomial coefficients	$O(2^n)$	$O(n * k)$
longest common subsequence	$O(2^{n+m})$	$O(n * m)$
mtx-chain multiplication	$O(n * 3^n)$	$O(n^3)$
string editing distance	$O(3^{n+m})$	$O(n * m)$
dag path sequence	$O(2^n)$	$O(n^2)$
optimal polygon triangulation	$O(n * 3^n)$	$O(n^3)$
optimal binary search trees	$O(n * 3^n)$	$O(n^3)$
paragraph formatting	$O(n * 2^n)$	$O(n^2)$
paragraph formatting 2	$O(n * 2^n)$	$O(n * width)$

[LS99]: *European Symp. Programming.* [LS03]: *Higher-Order & Symbolic Computation.*

Recursion. Use Indexed Structures Too

examples	original program's running time	opt'd prog's running time
Fibonacci function	$O(2^n)$	$O(n)$
binomial coefficients	$O(2^n)$	$O(n*k)$
longest common subsequence	$O(2^{n+m})$	$O(n*m)$
matrix-chain multiplication	$O(n*3^n)$	$O(n^3)$
string editing distance	$O(3^{n+m})$	$O(n*m)$
dag path sequence	$O(2^n)$	$O(n^2)$
optimal polygon triangulation	$O(n*3^n)$	$O(n^3)$
optimal binary search tree	$O(n*3^n)$	$O(n^3)$
paragraph formatting	$O(n*2^n)$	$O(n^2)$
paragraph formatting 2	$O(n*2^n)$	$O(n*width)$
0-1 knapsack	$O(2^n)$	$O(n*weight)$
context-free-grammar parsing	$O(n*(2*size+1)^n)$	$O(n^3*size)$
single-source shortest paths	$O(n^{n+1})$	$O(n*e)$
single-pair shortest paths	$O(n^n)$	$O(n*e)$
all-pairs shortest paths	$O(n^2*3^n)$	$O(n^3)$

[LS02]: *ACM SIGPLAN PEPM.*

Recursion. Transform to Iteration

- incrementalization produces iterative computation and facilitates additional optimizations, especially of heap
- reduce stack space and heap space
- examples:

  factorial:

```
fac(n) =
  if n=0 then 1
  else n*fac(n-1)
```

4∗(3∗(2∗(1∗1)))

```
fac1(n) = fact1(n,1)
fact1(n,r) =
  if n=0 then r
  else fact1(n-1,n*r)
```

bad: 1∗(2∗(3∗(4∗1)))

```
fac2(n) = fact2(n,0,1);
fact2(n,i,r) =
  if i=n then r
  else fact2(n,i+1,(i+1)*r)
```

good: 4∗(3∗(2∗(1∗1)))

  selection sort (update arg. in place):

	sort	least	rest
time	30 times faster	10 times faster	25 times faster
space	$O(n^2) \rightarrow O(n)$	$O(n) \rightarrow O(1)$	$O(n) \rightarrow O(1)$

[LS00]: *ACM SIGPLAN PEPM.*

Sets. Finite Differencing by Bob Paige

- expensive set expressions in loop body $\longrightarrow$ f

  e.g., $E = \{x \in S \,|\, p(x)\}$

  updates to parameters of such expressions $\longrightarrow$ $\oplus$

  e.g., $S\ with := a$

- exploit algebraic properties of set operations

  e.g., **if** $p(a)$ **then** $E\ with := a$

- extremely powerful at the set level:

  + dominated convergence at higher level

  + real-time simulation at lower level

Fixed-point exp.'s $\xrightarrow{DC}$ $\xrightarrow{RTS}$ pointer-based impl.

while-loops $\xrightarrow{FD}$ inc. set operations

Sets. Finite Differencing Examples

straightforward programs take higher-order polynomial time.

graph reachability cycle testing attribute closure	start with fixed-point exp's
topological sort tree center maximal independent set	start with loops w/set op's
DFA minimization ready simulation tree pattern matching	improved algorithms
relational calculus query optimization model checking pointer alias analysis	in Goyal's dissertation
grammar-constraints simplification parametric regular path queries implementation of Datalog rules	more recent results of ours

Rules. Implement Relations and Rules

trans. clo. example: `edge(u,v)` $\rightarrow$ `path(u,v)`
`edge(u,w)` $\wedge$ `path(w,v)` $\rightarrow$ `path(u,v)`

previous results:

large interpretive overhead; no tight worst-case guarantee; time sensitive to order of rules & hypotheses; space less known.

w/o memo: nonterminating or exponential time, depending on rule order.

with memo: `#edge*#vert` or ...`*#vert` time, depending on hypothesis order.

our results: overcome these (automatically generate specialized programs with precise time and space guarantees), plus

optimal time: once for each useful combination of facts.
optimal assoc space: minimum for time modulo scheduling.
providing output space and auxiliary space separately.

time: $O($`#edge`$\times$`#vertex`$)$, output space: $O($`#vertex`$^2)$, aux space: $O($`#edge`$)$.

[LS03]: *ACM SIGPLAN Intl. Conf. Principles and Practice of Declarative Programming.*

Rules: Implementing Datalog Examples

worst-case complexities (O):

problem	running time	output space	auxiliary space
transitive closure	`#edge`×`#vertex`	`#vertex`^{2}	`#edge`
graph reachability	`#source`+`#edge`	`#vertex`	1
existential and universal regular path queries	`#edge`×`#state`+ `#vertex`×`#trans.`	`#vertex`× `#state`	`#label`× `#vertex`×`#state`
existential and universal parametric RPQ	above × `#subst`×`#param`	above × `#subst`	above × `#subst`
constraint simplification	`#node`^{3}	`#node`^{2}	`#node`^{2}

reach: time `#source+min(#edge,#reach`×`#edge.2/1)`

RPQ: time `#edge`×`#trans`, space `#label`×`(#vertex`+`#state)` [LY-MPC02]

constraints: time `#simp`×`min(k+#copy.1/2,k*#simp.{2..k+2}/1)` [LLS-SAS01]

other applications:

mobile ambients nesting analysis [Braghin et al-VMCAI03]

context-free lang reach, model checking pushdown systems

...

Summary

incrementalization
- use old result, intermediate results, and auxiliary information

III: Iterate, Incrementalize, Implement
- optimize loops, recursion, sets, and rules

applications to problems in many domains
- list processing, graph algs, VLSI design, image processing, DB query,...

prototype: system CACHET and extensions
- fast prototyping, efficient implementation, good interface, ...

analysis of dependency on recursive data, and time and space
- use appropriate abstractions; automatic, precise, and efficient

Conclusion

- incrementalization is important for efficiency improvement.
  - III is important for efficiency guarantees.

- we have a general and systematic approach.
  - fully-automatic for optimizing compilers
  - semi-automatic in transformational programming
  - manually as a methodology for algorithm derivation

- our ultimate goal is to build supporting tools.:
  - convenient, efficient, & powerful prog analyses & transform.

- future work:
  - efficiency, effectiveness, additional language features
  - application: specialized methods, domain-specific knowledge
  - implementation: extend and improve CACHET