From Recursion to Iteration: What Are the Optimizations?

Y. Annie Liu and Scott D. Stoller

Computer Science Department
Indiana University

Recursion vs. Iteration

Recursion: `fac(n) { return if n==0 then 1 else n*fac(n-1); }`

```
fac(4)                                                    24
 \_4* fac(3)                                        4*6=24_/
      \_3* fac(2)                              3*2=6_/
           \_2* fac(1)                    2*1=2_/
                \_1* fac(0)  1*1=1_/                  need to
                     \_  1  _/                     keep stack
```

Iteration:
```
fac1(n) {return fact1(n,1);}
fact1(n,r) {return if n==0 then r else fact1(n-1,n*r);}
```

```
fac1(4)= fact1(4,1)    4*1                                24
           \_fact1(3,4)    3*4                           _/
                \_fact1(2,12)   2*12                  _/
                     \_fact1(1,24)   1*24          _/
                          \_fact1(0,24)   _/     can just
                               \_  24  _/    update store
```

fac → fac1:

Is it an optimization? No, in terms of time. Yes, in terms of space.

From Recursion to Iteration

Recursion:
```
fac(n) { return
  if n==0 then 1
  else n*fac(n-1); }
```
4∗(3∗(2∗(1∗1)))

$\not\to$

```
fac1(n) { return fact1(n,1); }
fact1(n,r) { return
  if n==0 then r
  else fact1(n-1,n*r); }
```
(1∗(2∗(3∗(4∗1))))

↓

Iteration:
```
fac2(n) {return fact2(n,0,1);}
fact2(n,i,r) { return
  if i==n then r
  else fact2(n,i+1,(i+1)*r); }
```
tail recursion

```
fac3(n) {
  i=0; r=1;
  while (i!=n) {i=i+1; r=i*r;}
  return r; }
```
loop

fac → fac2 or fac3:

Is it an optimization? Yes, in terms of both time and space.

What Are the Optimizations?

Running times (ms) and space usage of factorial in Scheme:

input n	fac(n)	fac1(n)	fac2(n)	fac3(n)
2000	176	182	160	158
4000	722	728	649	636
6000	1558	1688	1468	1438
8000	2862	3050	2716	2642
space	$O(n^2 \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Note: factor of $n \log n$ arises because $\log(n!)$ is $O(n \log n)$.

Q: How to obtain fac2 or fac3 from fac systematically?
What are the optimizations and how to guarantee?

Transforming Recursion into Iteration

Based on *incrementalization*: **different from previous work**

— essentially does same computation as the original program, except in an iterative and incremental fashion, avoiding unnecessary computations along the way.

- General:
  is not based on particular schemas or properties of programs
- Powerful:
  yields nontrivial opt (shorter, faster prog), succeeds in most cases
- Simple:
  has 3 intuitive steps

Language

e ::=	v		variable
|	c(e1,...,en)		data construction
|	p(e1,...,en)		primitive operation
|	if e1 then e2 else e3		conditional expression
|	let v=e1 in e2 end		binding expression
|	f(e1,...,en)		function application
s ::=	return e;		return statement
|	break;		break statement
|	v=e;		assignment
|	s1 s2		sequential statement
|	if (e) {s1} else {s2}		conditional statement
|	while (e) {s1}		loop statement

program: a set of f(v1,...,vn) {s}
s: a return stmt or a seq stmt that ends with a return stmt

Base Cases and Recursive Cases

Base case: **a condition under which f is not called recursively and returns a uniquely determined value**

Recursive case: **a condition under which f is called recursively and the rec call(s) have uniquely determined argument(s)**

```
f(n) {return if n≤0 then 0 else g(n);}
g(n) {return if n≤10 then n else f(n-5);}
```

f base: [$n \leq 0$]
[$n \not\leq 0, n \leq 10$]
f rec: [$n \not\leq 0, n \not\leq 10$]

Properly defined: **all base cases and rec cases of f appear in f alone**

Most funs are or easily made p.d.: f (and g) above? N f (and h) below? Y

```
f(n) {return if n≤0 then 0 else if n≤10 then n else h(n);}
h(n) {return f(n-5);}
```

Transform one fun at a time: start with main fun, and repeatedly process funs called in resulting fun.

Outline

0. 1 simple base case, 1 rec case, 1 rec call — **basic approach**

1. multi base cases (incl 1 base case not simple) — **generalize 0**

2. multi rec cases (but 1 rec call in each case) — **generalize 1**

3. rec on linear rec data structures — **additional optimizations**

4. multi rec calls (in 1 rec case) — **look hard, but orthogonal**

Separate issues shows explicitly why each technique is needed.

0. Basic Approach

```
f(x) { return if x==x0 then b(x0) else a(x,f(d(x))); }
```

1 base case
1 rec case
1 rec call

Step 1. Identify an increment op $\oplus$ to argument of f: $x' = x \oplus y$ s.t. $x = prev(x')$: $prev(x)$ is based on arguments of rec calls.

$prev(x) = d(x)$; $x \oplus y = d^{-1}(x)$ if d^{-1} exists, y: dummy val _

Step 2. Derive an incremental prog $f'(x, r)$ that computes $f(x)$ efficiently using the result r of $f(prev(x))$.

$f'(x, r) = a(x, r)$

Step 3. Form an iterative prog that initializes using base case of f and iterates using $\oplus$ and f'.

```
f(x) { x.=x0; r=b(x0);                                   init
       while (x.!=x) { x.=d^-1(x.); r = a(x.,r); }       iterate
       return r; }
```


0.1. Associativity

```
f(x) { return if x==x0 then b(x0) else a(x,f(d(x))); }
```

```
f(x) { x.=x0; r=b(x0);
       while (x.!=x) { x.=d^-1(x.);  r = a(x.,r); }
       return r; }
```

if a(x,y) is a1(a2(x),y), a1 is assoc & has id, could transform to:

```
f(x) { r=id;
       while (x!=x0) { r=a1(r,a2(x));  x=d(x); }
       return a1(r,b(x0)); }
```

– reverse direction of iteration: need no temp var x., or d^{-1}
– basically a1(a2(x.),r) $\rightarrow$ a1(r,a2(x)): check whether faster

e.g., fac: n$*$ ((n-1)$*$(n-2)) $\rightarrow$ (n$*$(n-1)) $*$(n-2) **maybe slower**
e.g., reverse: app(app(x,y),z) $\rightarrow$ app(x,app(y,z)) **clearly faster**

1. Multiple Base Cases

```
f(x) { return if c0(x) then b0(x)
              else if c1(x) then b1(x) else if ...     } multiple
              else a(x,f(d(x))); }                      } base cases
```

Steps 1 and 2: As before.   Step 3: Use an initial loop:

```
f(x) { x.=x;
       while (true)                                              } initial
         if (c0(x.)) { r=b0(x.); break; }                        } loop
         else if (c1(x.)) { r=b1(x.); break; } else if ...
         else x.=d(x.);
       while (x.!=x) { x.=d^-1(x.); r=a(x.,r); }
       return r; }
```

Not schema based: based only on base cases and recursive cases.

2. Multiple Recursive Cases

```
f(x) { return if c0(x) then b0(x)
              else if c1(x) then b1(x) else if ...
              else if t0(x) then a0(x,f(d0(x)))                  } multiple
              else if t1(x) then a1(x,f(d1(x))) else ...;}       } rec cases
```

Steps 1 and 2: As before.   Step 3: Use a stack:

```
f(x) { x.=x; s=nil;                                          init stack
       while (true)
         if (c0(x.)) { r=b0(x.); break; }
         else if (c1(x.)) { r=b1(x.); break; } else if ...
         else { s=cons(x.,s);                                push
                if (t0(x.)) x.=d0(x.);                       }
                else if (t1(x.)) x.=d1(x.); else ... }       }
       while (x.!=x) {
         x.=car(s); s=cdr(s);                                pop
         if (t0(x.)) r=a0(x.,r);                             }
         else if (t1(x.)) r=a1(x.,r); else ... }             }
       return r; }
```

3. Recursion on Linear Recursive Data Structures

```
sqrlist(x) { return if null(x) then nil
                    else cons(car(x)*car(x), sqrlist(cdr(x))); }
```

Steps 1-3: As before.
1. $prev(x) = cdr(x)$; $x' = x \oplus y = cons(y, x)$, $y = car(x'), x = cdr(x')$
2. $sqrlist'(x, r) = cons(car(x) * car(x), r)$, for $r = sqrlist(cdr(x))$

```
sqrlist0(x) { x.=x; s=nil;                                          init stack
              while (true) if (null(x.)) { r=nil; break; }
                           else { s=cons(x.,s); x.=cdr(x.); }       push
              while (x.!=x) { x.=car(s); s=cdr(s);                  pop
                              r=cons(car(x.)*car(x.),r); }          sqrlist'
              return r; }
```

Additional optimization:
1. Elim stack allocation using pointer reversal: ensure speedup.
2. Elim stack & backtracking on data construction: a bit faster.
3. Elim heap allocation using in-place update: drastic speedup.

3.1. Eliminating Stack Allocation

```
sqrlist0(x) { x.=x; s=nil;
              while (true) if (null(x.)) { r=nil; break; }
                           else { s=cons(x.,s); x.=cdr(x.); }       push
              while (x.!=x) { x.=car(s); s=cdr(s);                  pop
                              r=cons(car(x.)*car(x.),r); }
              return r; }
```

push: s=cons(x.,s) $\rightarrow$ cdr(x.)=s and s=x.
must keep cdr(x.) to do x.=cdr(x.) that follows:

```
s=cons(x.,s);   →  tmp=cdr(x.); cdr(x.)=s; s=x.;
x.=cdr(x.);     →  x.=tmp;
```

pop: x.=car(s) $\rightarrow$ cdr(s)=x. and x.=s
must keep $cdr(s)$ to do $s = cdr(s)$ that follows:

```
x.=car(s);      →  tmp=cdr(s); cdr(s)=x.; x.=s;
s=cdr(s);       →  s=tmp;.
```

sqrlist1(x): as sqrlist0 except with the above replacements.

3.2. Eliminating Stack and Backtracking

```
sqrlist0(x) { x.=x; s=nil;
              while (true) if (null(x.)) { r=nil; break; }
                           else { s=cons(x.,s); x.=cdr(x.); }       push
              while (x.!=x) { x.=car(s); s=cdr(s);                  pop
                              r=cons(car(x.)*car(x.),r); }          f'
              return r; }
```

if r is used only as components of constructed return value:

```
sqrlist2(x) { x.=x; r.=&r;                                          init
              while (true)
                if (null(x.)) { *r.=nil; break; }
                else { *r.=cons(car(x.)*car(x.), _);
                       r.=&cdr(*r.); x.=cdr(x.); }
              return r; }
```

– eliminate stack operations (the 4 assignments involving s)
– do work in first loop: move f' (*r=,r.=) to point of push

3.3. Eliminating Heap Allocation

```
sqrlist2(x) { x.=x; r.=&r;
              while (true)
                if (null(x.)) { *r.=nil; break; }
                else { *r.=cons(car(x.)*car(x.), _);
                       r.=&cdr(*r.); x.=cdr(x.); }
              return r; }
```

if all uses of x are selections of components, and same constructor is needed for the return value:

```
sqrlist3(x) { x.=x; r.=&r;
              while (true)
                if (null(x.)) { *r.=nil; break; }
                else { *r.=x.; car(x.)=car(x.)*car(x.);             diff
                       r.=&cdr(*r.); x.=cdr(x.); }
              return r; }
```

– Note: argument is modified

3.4. Recursion on Linear Recursive Data Structures

Running times (ms) and space usage of sqrlist in C:

input size n	sqrlist	sqrlist0	sqrlist1	sqrlist2	sqrlist3
200	0.35	0.44	0.25	0.22	0.02
600	1.01	1.36	0.74	0.69	0.06
1000	1.71	2.26	1.24	1.16	0.10
space	O(n)	O(n)	O(n)	O(n)	O(1)
		slower	faster	a bit faster	much faster
			O(1) add'l space	O(1) add'l space	update in place

More Examples

```
sort(x) { return if null(x) then nil
                 else let v = least(x) in
                      cons(v,sort(rest(v,x))); }
least(x) { return if null(cdr(x)) then car(x)
                  else let v = least(cdr(x))
                       in if car(x)<v then car(x) else v; }
rest(i,x) { return if i==car(x) then cdr(x)
                   else cons(car(x), rest(i,cdr(x))); }
```

Try to write iterative versions yourself, much more difficult !

Running times (ms) and space usage in C: so much faster !

input size n	sort	sort3	least	least3	rest	rest3
200	41	1.49	0.1	0.02	0.32	0.01
600	379	11.63	0.32	0.02	0.97	0.03
1000	1109	31.48	0.55	0.04	1.64	0.09
space	O(n^2)	O(n)	O(n)	O(1)	O(n)	O(1)

4. Multiple Recursive Calls

```
fib(n) { return if (n==0) then 1;
                else if (n==1) then 1;
                else fib(n-1) + fib(n-2); }
```

Steps 1-3: As above, except:
Step 1: minimal input change in arguments of all rec calls, so x-1
Step 2: inc prog may use intermed result & aux info of $f(prev(x))$

```
fib~(n) { return if n==0 then tuple(1)
                 else if n==1 then tuple(1)
                 else let v=fib(n-1) in tuple(v+fib(n-2), v); }
fib~'(n,r~) { return if n==0 then tuple(1)
                     else if n==1 then tuple(1)
                     else if n==2 then tuple(2,1)
                     else tuple(1st(r~)+2nd(r~), 1st(r~)); }
```


4.1. Multiple Recursive Calls (cont'd)

```
fib1(n) { i=0; r~=tuple(1);
          while (i!=n) { i=i+1; r~=fib~'(i,r~); }
          return 1st(r~); }
```

Additional optimizations: 1. put base cases of fib~' out of loop.
2. reuse tuple in loop.

```
fib3(n) { if(n==0){i=0;a=1;} else if(n==1){i=1;a=1;} else{i=2;a=2;b=1;}
          while (i!=n) { i=i+1; a.=a; b.=b; a=a.+b.; b=a.; }
          return a; }
```

Running times (ms) and space usage in C:

input n	fib(n)	fib1(n)	fib2(n)	fib3(n)
40	26910	1.11	0.98	0.03
80	>200000	2.21	2.02	0.05
120	>200000	3.36	3.11	0.08
space	O(n^2)	O(n^2)	O(n^2)	O(n)

Summary

Transform recursion into iteration: based on **incrementalization**

one function at a time —

$$\left.\begin{array}{l}\textbf{0.}\ \text{identify base cases and rec cases}\\ \text{for each rec case}\left\{\begin{array}{l}\textbf{1.}\ \text{identify arg increment}\\ \textbf{2.}\ \text{derive incremental prog}\end{array}\right.\\ \textbf{3.}\ \text{form iteration, with additional opts}\end{array}\right\}\begin{array}{l}\text{general}\\ \text{powerful}\\ \text{simple}\end{array}$$

Ongoing and future work:

- recursion on tree-structured recursive data
- implementation
- automatic time and space (improvement) analysis
- theoretical study of complexity improvements