From recursion to iteration: what are the optimizations?

Yanhong A. Liu* and Scott D. Stoller*

July 1999

Abstract

Transforming recursion into iteration eliminates the use of stack frames during program execution. It has been studied extensively. This paper describes a general and powerful method, based on incrementalization, for transforming general recursion into iteration: identify an input increment, derive an incremental version under the input increment, and form an iterative computation using the incremental version. Exploiting incrementalization yields iterative computation in a uniform way and also allows additional optimizations to be explored cleanly and applied systematically, in most cases yielding iterative programs that use constant additional space, reducing additional space usage asymptotically. We summarize major optimizations, complexity improvements, and performance measurements. Our analyses and measurements show that some previously considered "optimizations" can actually result in slower programs.

1 Introduction

Recursion refers to computations where the execution of a function or procedure calls itself and proceeds in a stack fashion. Iteration refers to repeated execution of a piece of code by explicitly updating a store and performing jumps. Transforming recursion into iteration eliminates the use of stack frames during program execution. This eliminates the space consumed by the stack frames as well as the time overhead of allocating and deallocating the frames, yielding significant performance improvement in both time and space. In terms of time, this may be a significant constant factor, especially if a program consists of small functions. In terms of space, the saving may be asymptotic.

While recursion is usually coded as recursive functions, iteration is usually programmed as loops. It is well-known that iteration corresponds to tail recursion, which is a recursion that performs no computation after the recursive call returns and thus may be implemented by copying the arguments and then performing a jump. Some programming language specifications, such as Scheme [1], require this implementation. Other languages consider this a compiler optimization. For example, this is apparently not always safe in Java for security reasons [18] and the current JVM apprently does not have instructions needed for tail recursion. Regardless of how tail recursion is implemented, it is just a special case of recursion. It has remained extremely challenging to develop general and powerful methods for transforming general recursion into iteration (loop or tail recursion), even though this is widely studied, as discussed at the end.

*This work is supported in part by NSF under grant CCR-9711253 and ONR under grants N00014-99-1-0132 and N00014-99-1-0358. Authors' address: Computer Science Department, Lindley Hall 215, Indiana University, Bloomington, IN 47405. Email: {liu,stoller}@cs.indiana.edu.

A little surprise. Factorial function is probably the most widely-used example for illustrating and comparing recursion and iteration, e.g., see [41, 16]. Function $fac(n)$, shown below, where $n \geq 0$, directly corresponds to the mathematical definition of factorial; it recurses in a stack fashion, computes multiplications when recursions return, and takes linear stack space. Function $fac1(n)$ computes the same factorial function in an iterative (tail recursive) fashion; it uses a second parameter to accumulate the result of multiplications, does nothing when recursions return, and may take constant space. Using an implementation with tail-recursion optimization, $fac1(n)$ is expected to run much faster than $fac(n)$, since the overhead of stack allocation and deallocation in $fac(n)$ is completely eliminated.

$$fac(n) \triangleq \begin{array}{l}\textbf{if } n = 0 \textbf{ then } 1 \\ \textbf{else } n * fac(n-1)\end{array} \qquad \begin{array}{l} fac1(n) \triangleq fact1(n, 1) \\ fact1(n, r) \triangleq \begin{array}{l}\textbf{if } n = 0 \textbf{ then } r \\ \textbf{else } fact1(n-1,\ n * r)\end{array}\end{array}$$

To our surprise, measuring the running time of $fac(n)$ and $fac1(n)$ as programs written in Scheme, we found that the latter is slower than the former, as shown in Figure 1. Why? Notice that $fac(n)$ computes $n{*}((n{-}1){*}(...{*}(3{*}(2{*}1))...))$, while $fac1(n)$ computes $(((...(n{*}(n{-}1)){*}...){*}3){*}2){*}1$. Even though $fac(n)$ and $fac1(n)$ perform an equal number of multiplications, the former performs them on smaller numbers than the latter. Since the result of factorial may be quite large, arbitrary precision multiplication is used, and thus multiplying smaller numbers are faster than multiplying larger numbers. This differences outweighs the savings of the tail-recursion optimization. To confirm this, we use two new programs sum and $sum1$ that differ from fac and $fac1$, respectively, only by replacing $n*$ with $1+$. Indeed, the tail-recursive version is faster.

To summarize, $fac1$ reverses the order of the numbers to be multiplied. Deriving $fac1$ from fac requires exploiting the associativity of multiplication and produces a slower program.

From recursion to iteration. Our method for transforming recursion into iteration, when applied to program fac, produces the following program $fac2$, which uses a third argument i in the middle that grows from 0 to n:

$$\begin{array}{l} fac2(n) \triangleq fact2(n, 0, 1) \\ fact2(n, i, r) \triangleq \begin{array}{l}\textbf{if } i = n \textbf{ then } r \\ \textbf{else } fact2(n,\ i+1,\ (i+1) * r)\end{array}\end{array}$$

As shown in Figure 1, $fac2(n)$ is faster than $fac(n)$. The optimized program $fac2$ performs the same multiplications as fac. No algebraic properties of multiplication are used. The optimization simply eliminates the stack and guarantees performance improvement.

As an additional note, applying our optimization to program sum, we obtain $sum2$ that uses three parameters and is indeed slightly slower than $sum1$. However, both $sum1$ and $sum2$ use constant space and are much faster than sum. We consider it a separate optimization to reduce the three arguments of $sum2$ to the two arguments of $sum1$ by exploiting associativity.

Even though tail recursion may be implemented efficiently as loops, loops are more widely supported in compilers for generating efficient code. Therefore, we transform recursive functions into loops directly. Note that generating tail recursion merely requires different syntax. Figure 1 shows the measurements for $fac3$, the version of $fac2$ using loop. Even though there is a small difference between the running times of $fac2$ and $fac3$, we think it is small enough to be a random artifact. The $O(n \log n)$ factor for space usage is due to the use of arbitrary-precision integers.

Our transformation is based on incrementalization [37, 35, 36]. The method consists of three steps: (1) identify an input increment, (2) derive an incremental version under the input increment, and (3) form an iterative computation using the incremental version. It applies uniformly to

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

Figure 1: Running times (in milliseconds) and space usage of factorial in Scheme.

recursive functions that are non-linear, mutually recursive, and use recursive data structures. We present the transformation by considering increasingly more general forms: recursions with multiple base cases, with multiple recursive cases, and with multiple recursive calls in one recursive case, i.e., non-linear. We also describe additional optimizations on recursive data structures. The method has been applied to all examples we found in the literature, addressed previously using a large variety of different methods, and succeeded in transforming all of them into iteration. In many cases, we obtain the resulting program in much fewer steps and obtain shorter programs, some not found previously. Our method guarantees performance improvements and is fully automatable. All performance measurements shown in this paper are performed on a Sun Ultra 10 with 300MHz and 124MB memory. Those in Figure 1 used the Chez Scheme compiler. The others used the gcc compiler; we also measured some of them in Java 1.1 and obtained similar speedups.

The rest of the paper is organized as follows. Section 2 describes the language and basic concepts. Section 3 describes the basic idea using recursion with one simple base case and one recursive case with one recursion call. Section 4 handles multiple base cases, including cases with ranges. Section 5 addresses multiple recursive cases, each with one recursive call. Section 6 discusses recursion on data structures, with additional optimizations that eliminate stack allocation and heap allocation. Section 7 handles multiple recursive calls in a recursive case, i.e., non-linear recursion. Section 8 summarizes the entire algorithm and discusses related issues. Section 9 compares with related work.

2 Preliminaries

We use a simple programming language with the following grammar for expressions and statements:

$e ::= v$	variable	$s ::=$ **return** e;	return statement
$\mid\ c(e_1, ..., e_n)$	data construction	$\mid$ **break** ;	break statement
$\mid\ p(e_1, ..., e_n)$	primitive operation	$\mid\ v = e;$	assignment
$\mid$ **if** e_1 **then** e_2 **else** e_3	conditional expression	$\mid\ s_1\ s_2$	sequential statement
$\mid$ **let** $v = e_1$ **in** e_2 **end**	binding expression	$\mid$ **if** (e) $\{s_1\}$ **else** $\{s_2\}$	conditional statement
$\mid\ f(e_1, ..., e_n)$	function application	$\mid$ **while** (e) $\{s_1\}$	loop statement

A program is a set of function definitions of the form $f(v_1, ..., v_n)$ $\{s\}$, where s ends with a **return** statement, and a function f_0 that is to be evaluated with some input. We use a strict semantics, so this language contains an untyped core subset of many programming languages, including Java, C, ML, and Scheme. We use Java and C syntax for primitive operations on Booleans and numbers and for statements. Braces enclosing a non-sequential statement can be omitted. We use Lisp and Scheme syntax for data construction with constructors *nil*, *cons*, and so on and for component selection with selectors *car*, *cdr*, and so on; a constant is simply a constructor of arity 0, and we write $c()$ as c. We use ML syntax for conditional and binding expressions. For easy of type setting, we use typewriter font for standalone code below.

In this paper, we assume that the body of each given function is a **return** statement. Other forms of statements are used in the optimized program. Additionally, when transforming recursion on recursive data structures, optimizations use destructive update, i.e., assignments to components of compound values (such as $car(x) = y$). Furthermore, allowing taking addresses ($x = \&y$) and storing values in addresses ($*x = y$) yield even clearer and more efficient code, as discussed in Section 6.

Properly-defined functions. We introduce important definitions that capture when a given function is *properly-defined*. Given a function f, a *static path* in f is a sequence of conditions, or their negations, in the definition of f that are evaluated in order. A *base case* of f is a maximal static path in f following which f will not be called. A *recursive case* is a minimal static path following which f will be called. For example, static path $n == 0$ in fac is the only base case of fac, and $!(n == 0)$ is the only recursive case. A static path of f might not be a base case or a recursive case of f. For example, for function f below, the static path $!(n <= 0)$ is neither a base case nor a recursive case of f.

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

We say that f is *properly-defined* if all static paths of f are base cases or recursive cases. So, fac is properly-defined, and f and g are not. Note that mutually recursive functions can be properly-defined. For example, if function g above were changed to only `g(n) { return f(n-5); }`, then f and g would be mutually recursive and properly-defined. Note that most functions that arise naturally are properly-defined. This notion allows us to simplify the presentation of our method.

The method we study in this paper applies to all properly-defined functions, regardless of whether they are mutually recursive or non-linear. We start with transforming f_0, and repeatedly transform functions that are called in the resulting program. As a special case, if all static paths in a function are recursive cases, then we immediately conclude that the function does not terminate correctly and do not transform it. Of course, if all static paths in a function are base cases, then we do not need to transform it either.

For ease of analysis and transformation, we assume that a preprocessor gives a distinct name to each bound variable and lifts bindings so that they are not the arguments of data constructions, primitive operations, or function applications, and are not in the condition or binding positions [36]. For example, $cons(f(x),$ **let** $v = e_1$ **in** $e_2)$ becomes **let** $v = e_1$ **in** $cons(f(x), e_2)$, and **if** (**let** $v = e_1$ **in** e_2 **end**) **then** $g(x)$ **else** $h(x)$ becomes **let** $v = e_1$ **in** **if** e_2 **then** $g(x)$ **else** $h(x)$ **end**. After this preprocessing, we can easily transform the body of a given function into statements that contain no binding expressions by repeatedly transforming **return let** $v = e_1$ **in** e_2 **end**; into $v = e1$;**return** e_2;. This allows us to produce resulting programs directly in C or Java syntax, which do not have binding expressions.

For ease of forming initialization code based on the base cases, for base cases, preprocessing also lifts conditions so that they are not the arguments of data constructions, primitive operations, or function applications, and are not in the condition or binding positions [36]. For example, $cons(f(x),$ **if** e_1 **then** e_2 **else** $e_3)$ becomes **if** e_1 **then** $cons(f(x), e_2)$ **else** $cons(f(x), e_3)$. After this, we can easily transform the base cases into statements that contain no conditional expressions by repeatedly transforming **return if** e_1 **then** e_2 **else** e_3; into **if** (e_1) **return** e_2; **else return** e_3;.

To simplify the presentation, we describe the transformation for functions with one argument. However, our general methods can be extended easily to handle multiple arguments.

3 Basic approach

We consider in this section simple recursion with one base case and one recursive case with one recursive call. Factorial is a good example for illustrating the basic approach, which consists of three steps.

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

First, identify an input increment by analyzing the arguments of recursive calls. The single recursive call in fac says that factorial of n recursively computes factorial of $n-1$. The increment on which the computation can be performed is the inverse of change to the argument of the recursive call. So, the input increment is from $n-1$ to n, or equivalently from n to $n+1$; in the former view, $n-1 \geq 0$ (by definition of fac, as shown below), and in the latter, $n \geq 0$ (can be shown similarly).

Then, incrementalize the recursive computation by transforming the function on the incremented input to use the result of the function on the previous input. That is, transform $fac(n)$ to use the result of $fac(n-1)$, or equivalently transform $fac(n+1)$ to use the result of $fac(n)$. This paper takes the former of the two equivalent views.[1] This yields an incremental version fac', derived below. The syntax "**return** ...;" does not affect the derivation and thus is omitted.

$$\begin{array}{ll}
fac'(n,r), & \text{where } r = fac(n-1)\text{, which implies that } n-1 \geq 0\text{, by definition of } fac \\
= fac(n) & \text{this is the goal: } fac'(n,r) \text{ computes } fac(n) \text{ using } r \\
= \textbf{if } n{=}{=}0 \textbf{ then } 1 \textbf{ else } n * fac(n-1) & \text{by definition of } fac \\
= n * fac(n-1) & \text{by simplifications: } n{=}{=}0 \longrightarrow \mathit{false}\text{, since } n{-}1 \geq 0\text{, and } \textbf{if } \mathit{false}\ a\ b \longrightarrow b \\
= n * r & \text{by replacement: } fac(n-1) \longrightarrow r
\end{array}$$

General methods for deriving incremental programs are described in [37] and summarized in [36]. The simplification needed on primitive arithmetic and Boolean operations can be performed automatically and efficiently using systems like Omega [45] and MONA [30].

Finally, form an iterative computation by copying the base case and base value and, for the recursive case, iterating using the incremental version. For factorial, the base case is $n = 0$ and $fac(0) = 1$. We initialize the state with the base case and use i to iterate till n.

```
fac3(n) { i=0; r=1;
          while (i!=n) { i=i+1; r=fac'(i,r); }
          return r; }
```

At the end, `fac'(i,r)` is inlined, yielding the final optimized program.

The basic method. The basic method described above applies to any recursive function f on x with one simple base-case condition $x = x_0$ and base value $b(x_0)$, and with one recursive call in the only recursive case $a(x, f(d(x)))$:

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

Note that symbols a, b, and so on may denote any pieces of code, not necessarily functions, with unbound occurrences of their arguments; of course, they should not include other base cases, recursive cases, or recursive calls to f.

To transform recursive function f into iteration, three steps are performed. Step 1 identifies an increment $\oplus$ to the argument of f, i.e., $x' = x \oplus y$ such that $x = prev(x')$, where $prev(x)$

[1] This view is also taken in a recent paper [34] and is more direct for program optimization using incrementalization. The alternative view is taken in our previous papers and is more direct for incrementalization alone.

is based on the arguments of recursive calls. In this case, $prev(x) = d(x)$ and, if d^{-1} exists, $x \oplus y = d^{-1}(x)$, where any dummy value, denoted $\_$, can be plugged in for y. Step 2 derives an incremental program $f'(x,r)$ that computes $f(x)$ efficiently using the result r of $f(prev(x))$. In this case, $f'(x,r) = a(x,r)$. Step 3 forms an iterative program that initializes using the base case of f and iterates using f'. In this case, we set iterating variable $x. = x_0$ and result $r = b(x_0)$ and, as long as $x.$ is not equal to input x repeatedly set $x. = x. \oplus \_$ and $r = f'(x.,r)$:

```
f(x) { x. = x0; r = b(x0);
       while (x. != x) { x. = d_inverse(x.); r = a(x., r); }
       return r; }
```

For the simple recursion considered here, the optimization succeeds if Step 1 does, i.e., if d^{-1} exists. Finding function inverses is a separate topic of research [22]; our general algorithm does not rely on it, as described in Section 5, but uses it when it is available, as above. It is easy to prove that the optimized program terminates with a value exactly when the original program terminates with the same value. The optimized program avoids using the linear stack space that the original program does, and it runs much faster if d^{-1} is inexpensive. On a non-terminating input, the optimized program will be in an infinite loop while the original program will run out of stack space.

The fact that fac can be transformed into $fac3$ has been studied by many researchers. What's new here is the three-step method that is general and powerful, as shown in the following sections.

Exploiting Associativity. When $a(x,y)$ above is of the form $a_1(a_2(x),y)$ and a_1 is associative, then we could directly form the following iterative program, where no temporary variable $x.$ is needed:

```
f(x) { r=b(x0);
       while (x!=x0) { r=a_1(r,a_2(x)); x=d(x); }
       return r; }
```

Explicitly exploiting associativity allows us to use this transformed program only if this indeed yields speedups. For factorial, this amounts to replacing $n*((n-1)*(n-2))$ by $(n*(n-1))*(n-2)$, but the latter might be slower due to multiplying bigger numbers. For summation in Section 1, this amounts to replacing $1+(1+1)$ by $(1+1)+1$, which is equally fast, but removing the additional temporary variable will yield a speedup. For list reversal, this amounts to replacing $append(append(x,y),z)$ by $append(x,append(y,z))$, which clearly produces a speedup.

4 Multiple base cases

Consider a recursive function f on x with one or more base conditions $c_0(x)$, $c_1(x)$, ... and base values $b_0(x)$, $b_1(x)$, ..., respectively, and still with one recursive call in the only recursive case $a(x, f(d(x)))$, for example,

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

In general, f may have its base cases and recursive case interleaved, for example,

```
foo(x) { return if x>1 then if x<=50 then 4 else x*x+foo(x-7) else 20; }
```

The three steps described in Section 3 apply. Steps 1 and 2 remain exactly the same. The only problem for Step 3 is that, in general, we don't know which base case to use for initialization.

Using an initial loop. The general solution is to repeatedly decrement the input and test it against all base cases, just as the recursive calls would do. Basically, Step 3 forms an initial loop that decrements the input to the appropriate base case before the loop that computes the result incrementally:

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

The general algorithm for forming the initial loop (lines 2-4 above) has three steps. First, transform the function body into statements that contain no binding expressions, and transform the base cases into statements that contain no conditional expressions either, as described in Section 2. Then, replace the statement `return b(x);` that follows each base case with `{r=b(x.); break;}` and replace the statement `return a(x,f(d(x)));` that follows the recursive case with `{x.=d(x.);}`. Finally, use the resulting code as the body of a `while(true)` loop. For function *foo*, we obtain

```
while (true) if (x.>1) if (x.<=50) { r=4; break; } else { x.=x.-7; } else { r=20; break; }
```

The optimization preserves correctness and improves performance exactly as in Section 3.

We see that Section 3 handles a special case. It does not need the initial loop for testing and decrementing the input, because there is only one simple base case; every terminating computation must start at this base case.

Loop contraction on constant steps. On integer-valued arguments, if the decrement operation is the addition or subtraction of a constant, then optimizations can eliminate the first loop using a mod operation. We omit the details here.

5 Multiple recursive cases

Consider a recursive function f with one or more base cases, as in Section 4, and with multiple recursive cases, each containing one recursive call, for example,

```
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))) else if t1(x) then a1(x,f(d1(x))) else ...; }
```

Again, the three steps described in Section 3 apply. Steps 1 and 2 remain exactly the same, except that each recursive case gives rise to an input increment and an incremental version. The only additional problem for Step 3 is that, in general, for incrementing the parameter and computing the result incrementally, we don't know which input increment and incremental version to use in each iteration.

Using a stack. The general solution is to record information in a stack as the input is repeatedly decremented in the initial loop, and then to pop this information from the stack for the incremental computation in the second loop. As a result, Step 3 may yield the following iterative program where the information pushed and popped is the input parameter:

```
f(x) { x.=x; s=nil; //initialize stack, a singly linked list
       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 onto stack
                if (t0(x.)) x.=d0(x.); else if (t1(x.)) x.=d1(x.); else ... }
       while (x.!=x) {
         x.=car(s); s=cdr(s); //pop from stack
         if (t0(x.)) r=a0(x.,r); else if (t1(x.)) r=a1(x.,r); else ... }
       return r; }
```

In general, f may have its base cases and recursive cases interleaved. A general algorithm that handles this can be given in a similar fashion as in Section 4. In particular, all intermediate values that are computed before the recursive call are candidates to be pushed together with $x.$ on the stack; yet, if any of these values, including $x.$, is not used after the call returns, then it does not need to be pushed on the stack.

An important point here is that the transformation does not depend on the existence of d_i^{-1}. This allows recursive functions on recursive data structures to be handled directly; since additional optimizations can be done for recursion on recursive data structures, we describe them separately in Section 6. Essentially, we could transform any linear recursion into iteration without using stack frames. We will also see in Section 7 that issues arising from non-linear recursion are orthogonal. So, this is a most general iterative form.

The transformation preserves correctness exactly as in Section 3. It improves the running time if the times for allocating and deallocating the stack data structure are shorter than the times for allocating and deallocating stack frames; similarly for space usage. There may be a time-and-space trade-off, depending on the language, compiler, and architecture. For example, allocating and deallocating records in C might be faster than allocating and deallocating stack frames with some C compilers, though this is not the case in our experiment. This may or may not be true with Scheme or ML compilers when constructors like *cons* and function calls are both highly optimized. This is certainly not true with most Java compilers where object creation is very expensive. Therefore, this transformation should be applied only if it gives the desired performance improvements based on the language, compiler, and architecture. Most functions on numbers do have one recursive case and have an inverse for the decrement operation, so no stack is needed, as in Sections 3 and 4. All functions on recursive data structures can use additional optimization to remove the stack, as described in Section 6, and associativity can help further optimize many functions.

We see that Section 4 handles a special case. It does not need a stack, because that there is one recursive case and the decrement operation has an inverse, based on which the inputs that would have been kept in the stack can be constructed.

Eliminating redundant tests of recursive cases. A disadvantage of this program is that conditions t_i are tested both in the first and the second loops. To eliminate this redundancy, we can record the indices of the conditions also in the stack as they are tested in the first loop, and use them in the second loop. Clearly, this involves a time-and-space trade-off. The details are straightforward to work out.

6 Recursion on linear recursive data structures

We use function *sqrlist* that squares each element of a list as an example:

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

Again, the three steps apply, exactly as in Section 5 above. Step 1 obtains the decrement operation $prev(x) = cdr(x)$. Although cdr does not have an inverse, we may think of a corresponding increment operation $x' = x \oplus y = cons(y, x)$, so that given $y = car(x')$ as well as $x = cdr(x')$, we can obtain the incremented input $x' = x \oplus y$. Step 2 derives $sqrlist'$ that computes $sqrlist(x)$ using the result r of $sqrlist(prev(x))$: $sqrlist'(x, r) = cons(car(x) * car(x), r)$. Step 3 forms an iterative program that uses a stack to hold the input parameters:

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

The input data structure is not updated, so the equality test $!=$ can be done in constant time by pointer comparison.

Elimination of stack allocation using pointer reversal. We need to keep the stack, because cdr does not have an inverse, but we want to avoid allocating new space. This is possible with recursion on recursive data structures. The idea is to use the pointers in the input x, reversing the pointers as we go down the list and reversing them again as we go back up, as in DFS in mark-and-sweep garbage collection [3]. This is achieved by the following two changes to our algorithm.

In the first loop, we essentially want to achieve $s = cons(x., s)$ by setting $cdr(x.) = s$ and $s = x.$, but we must keep $cdr(x.)$ before it is reversed so that we can achieve $x. = cdr(x.)$ that follows. So, we use a temporary variable tmp to keep it, and then assign it to $x.$. The result is that we change $s = cons(x., s)$; to $tmp = cdr(x.); cdr(x.) = s; s = x.$; and change $x. = cdr(x.)$; to $x. = tmp$;. Note that s is still the stack.

In the second loop, we essentially want to achieve $x. = car(s)$ by setting $cdr(s) = x.$ and $x. = s$, but we must keep $cdr(s)$ before it is reversed so that we can achieve $s = cdr(s)$ that follows. So, we use a temporary variable tmp to keep it, and then assign it to s. The result is that we change $x. = car(s)$; to $tmp = cdr(s); cdr(s) = x.; x. = s$; and change $s = cdr(s)$; to $s = tmp$;. It is easy, and no surprise, to see the correspondence between this change and the one in the first loop. For $sqrlist$, we obtain the following iterative program:

```
sqrlist1(x) { x.=x; s=nil;
              while (true) if (null(x.)) { r=nil; break; }
                           else { tmp=cdr(x.); cdr(x.)=s; s=x.; x.=tmp; }
              while (x.!=x) { tmp=cdr(s); cdr(s)=x.; x.=s; s=tmp; r=cons(car(x.)*car(x.), r); }
              return r; }
```

We see that it is simple to achieve potentially complicated pointer reversal by transforming stack operations. The transformation is general and powerful, and it helps simplify programming with pointers significantly. Also, correctness of the resulting program is easily seen.

Reversing pointers is a curse when generational garbage collection is used. However, our further optimizations, as described below, help eliminate garbage creation and collection completely in many cases. We are also studying automatic space analysis. Our ultimate goal is to exactly predict and control memory allocation and deallocation, including the use of garbage collection.

Elimination of stack and backtracking on data construction. For incremental computation that uses the result r only as part of the data constructed, we can even eliminate the stack and backtracking, i.e., the second loop, completely. This requires that, in the incremental version obtained from Step 2, every subexpression that contains r be an argument of a data construction, a branch of a conditional expression, or the body of a binding expression. For example, the incremental computation may be

```
r=if (e0) triple(e1,r,let v=e3 in cons(r,e4)) else e5;
```

where none of the e_i contains occurrences of r. To eliminate the stack and backtracking, we modify the forward loop, i.e., the first loop, to keep addresses to plug in results from the rest of the computation.

In a lower-level language like C, this can be done easily and efficiently. In the forward loop, each iteration evaluates the right hand of the incremental computation to a data construction, using a dummy value for occurrence of r, stores it in each address in the list of addresses created in the previous iteration, and updates the list to contain the addresses (which currently contain dummy values) of the r's in the right hand side of the incremental version. Before the loop, the list of addresses contains only the address of r. The base value is assigned at the end of the loop. Now, we can remove the stack and stack operations, as well as the second loop. For *sqrlist*, there is only one address to be passed, so there is no need to create a list to hold it; it is simply held in $r.$. The resulting program is

```
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; }
```

It is trivial to actually reuse x for $x.$ and eliminate the first assignment; we kept $x.$ for easier comparison with previous versions of *sqrlist*.

In a higher-level language like Java, this transformation needs a level of indirection. We can't keep pointers to components of data constructions, i.e., fields of objects, so we must keep a pointer $r.$ to hold the enclosing data structure. This also requires the first iteration of the loop to be executed separately, since the pointer $r.$ could not be initialized before entering the loop. We obtain:

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

A more complicated general algorithm is needed compared with using C above; we need to keep pointers, components, as well as conditions under which appropriate assignments are done.

Elimination of heap allocation on data destruction. For data construction, reuse of heap space is desirable for highly efficient computation. This is an area of separate study, with a lot of existing work, e.g., see [24, 27, 51], but it has been difficult to design general and powerful methods. When forming iterative programs based on incrementalization, we can achieve reuse of heap space easily in a special but common case. In the incremental version, if all accesses to a given data construction $c(e_1, ..., e_n)$ are selections of its components, and a data construction of c is needed, then the given construction of c can be reused. We simply let the result refer to this data structure, and we assign the new fields directly in it. The optimized program updates its argument in-place, and therefore is usable only if there are no other pointers to the argument.

For *sqrlist* example, $x.$ is only referred to using $car(x.)$ and $cdr(x.)$, so the *cons* cell it corresponds to can be reused, i.e., the construction $*r = cons(car(x.) * car(x.), \_)$; can be changed to $*r = x.; car(x.) = car(x.) * car(x.)$.

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

This simple replacement completely eliminates heap allocation. Further analysis enables the elimination of $r.$.

input size n	*sqrlist*	*sqrlist*0	*sqrlist*1	*sqrlist*2	*sqrlist*3
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	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$

Figure 2: Running times (in milliseconds) and space usage of *sqrlist* in C

Examples. Function *sort* performs selection sort, selecting the minimum value to put at the beginning, and recursively sorting the rest of the list.

```
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))); }
```

We obtain the following functions, where we used elimination of stack and backtracking for *sort*3, elimination of stack allocation for *least*3, and elimination of stack, backtracking, and heap allocation for *rest*3. Figure 3 contains measurements of the running times.

```
sort3(x) { x.=x; r.=&r;
           while (true) if (null(x.)) { *r.=nil; break; }
                        else { v=least3(x.); *r.=cons(v,_); r.=&cdr(*r.); x.=rest3(v,x.); }
           return r; }

least3(x) { x.=x; s=nil;
            while (true) if (null(cdr(x.))) { r=car(x.); break; }
                         else { tmp=cdr(x.); cdr(x.)=s; s=x.; x.=tmp; }
            while (x.!=x) { tmp=cdr(s); cdr(s)=x.; x.=s; s=tmp; r=if (car(x.)<r) car(x.) else r; }
            return r; }

rest3(i,x) { x.=x; r.=&r;
             while (true) if (i==car(x.)) { *r.=cdr(x.); break; }
                          else { *r.=x.; r.=&cdr(*r.); x.=cdr(x.); }
             return r; }
```

In *sort*3, functions *least* and *rest* can be inlined, with renaming of local variables. For *least* we could form only a forward loop by exploiting the associativity of $a_1(x,y) = \textbf{if } x<y \textbf{ then } x \textbf{ else } y$; we used this version here to show the power of pointer reversal. Because *rest*3 modifies its input, so does *sort*3.

input size n	*sort*	*sort*3	*least*	*least*3	*rest*	*rest*3
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	$O(n^2)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$

Figure 3: Running times (in milliseconds) and space usage of sorting, etc., in C

7 Multiple recursive calls

Consider a general recursive function where a recursive case may have multiple recursive calls with different arguments. Fibonacci function is a simple example of this:

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

Once again, the three steps apply, except that Steps 1 and 2 need more general and powerful methods, and Step 3 may form a further optimized program.

For Step 1, note that an input increment operation should reflect how a computation may proceed in an incremental fashion. In general, a function may have multiple ways of proceeding, even if there is only one recursive call, e.g., proceeding at different step sizes. The idea is to analyze the arguments of all recursive calls and use a minimal input change. For Fibonacci, $prev(x) = x-1$ and $x \oplus y = x+1$.

A general algorithm for computing the increment operation is described in [34]. The basic ideas are to represent arguments of recursive calls so that the differences between them and x are explicit, and then extract minimal differences that cover all these arguments. The partial ordering on differences is: a difference involving fewer parameters is smaller; a difference in one parameter with smaller magnitude is smaller; other differences are incomparable. A set of differences covers a recursive call if the argument to the call can be obtained by repeated application of the given differences. For Fibonacci, the arguments of recursive calls are $x-1$ and $x-2$; trivially, $x-1$ gives rise to a smaller difference than $x-2$; and the argument $x-2$ can be obtained by applying $x-1$ twice. We have applied this simple method on all the examples we know, including all dynamic programming examples found in standard algorithm textbooks [34], and succeeded in all of them.

For Step 2, note that in the general problem of computing $f(x)$ using $f(prev(x))$, one may use not only the value of $f(prev(x))$ [37], but also intermediate results computed in $f(prev(x))$ [36] and auxiliary information not computed by $f(prev(x))$ at all [35]. This yields functions $\tilde{f}$ and $\tilde{f}'$, where $\tilde{f}(x)$ returns $f(x)$ tupled with other information needed for efficient incremental computation, and $\tilde{f}'(x,\tilde{r})$ computes $\tilde{f}(x)$ efficiently using the result $\tilde{r}$ of $\tilde{f}(prev(x))$. That is, if $f(x) = r$, then $1st(\tilde{f}(x)) = r$, where $1st$ retrieves the first component of a tuple, and if $\tilde{f}(prev(x)) = \tilde{r}$ and $f(x) = r'$, then $\tilde{f}'(x,\tilde{r}) = \tilde{f}(x)$ and $1st(\tilde{f}'(x,\tilde{r})) = r'$. Furthermore, the method of [37, 36, 35] ensures that $\tilde{f}'(x,\tilde{r})$ is at least as fast as $\tilde{f}(x)$. For Fibonacci, that method yields the following program [36], where *tuple* is a constructor of variable arity, and $1st$, $2nd$, and so on retrieve the corresponding components of a tuple.

```
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~)); }
```

General algorithms for obtaining $\tilde{f}$ and $\tilde{f}'$ that use information other than the return value of f are described in [36, 35]. The basic idea is to determine what to cache based on how cached values may be used in the incremental computation. The approach is to decompose an ambitious transformation into several simple transformations and dependence analyses. Since these include eliminating useless computations, which may be sources of nontermination, we may obtain incremental programs that terminate more often then the original program. Therefore, the general algorithm preserves semantics in the sense that, if the original program terminates with a value, then the incremental version terminates with the same value and computes asymptotically at least as fast.

For Step 3, we may form an iterative program as before, except that: the initialization is based on the base cases of $\tilde{f}$ rather than f; the loop body uses $\tilde{f}'$ rather than f' for the incremental computation; and $1st(\tilde{r})$ rather than r is returned. Actually, since the incremental version $\tilde{f}'$ handles base-case inputs as well, we can simply initialize using the smallest input and omit using the loop for decrementing the input; the decrementing loop is a little complicated and is of little practical benefit. For Fibonacci, this yields following program:

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

Note that $\widetilde{fib}'$ could be inlined.

Additional optimizations. Step 3 can be enhanced to form a further optimized program directly. Note that $\tilde{f}'(x,\tilde{r})$ computes $\tilde{f}(x)$ regardless of whether $\tilde{r}$ is available, i.e., $\widetilde{fib}'(n,\tilde{r})$ computes $\widetilde{fib}(n)$ for the base cases as well. However, when using $\tilde{f}'$ in a loop, the base cases are needed before the loop, not inside the loop. Thus, when forming an iterative program, the initialization can be based on the base cases of $\tilde{f}'$, and the loop body can use only the other cases of $\tilde{f}'$. For Fibonacci, this yields

```
fib2(n) { if(n==0) {i=0;r~=tuple(1);} else if(n==1) {i=1; r~=tuple(1);} else {i=2;r~=tuple(2,1);}
          while (i!=n) { i=i+1; r~=tuple(1st(r~)+2nd(r~),1st(r~)); }
          return 1st(r~); }
```

Another nice implication of optimization based on incrementalization is that we know that the space for $\tilde{r}$ can be reused from iteration to iteration, and thus no memory allocation or deallocation is needed in the loop. Static analysis can be used to determine the space and variables needed [32]. For Fibonacci, two variables, a and b, are used to hold the two components of $\tilde{r}$, where $1st(\tilde{r})$ is a, yielding

```
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; }
```

Note that even the optimized version $fib1$ gives drastic efficiency improvements over the original function; it takes linear time and requires constant space. In our previous work, which focuses on the derivation of $\tilde{f}$ and $\tilde{f}'$, we formed worse programs, which use recursion, not iteration, and still observed drastic speedup. There, a recursive case for $f(x)$ is **let** $r = f(prev(x))$ **in** $f'(x,r)$. For Fibonacci, we obtained the following two versions that correspond to $fib1$ and $fib2$ above:

```
fib1rec(n) { return 1st(fib1~(n)); }
fib1~(n) { return if n==0 then tuple(1)
                  else let r~=fib1~(n-1) in fib~'(n,r~); }

fib2rec(n) { return 1st(fib2~(n)); }
fib2~(n) { return if n==0 then tuple(1) else if n==1 then tuple(1)
                  else if n==2 then tuple(2,1)
                  else let r~=fib2~(n-1) in tuple(1st(r~)+2nd(r~),1st(r~)); }
```

Note that $fib1$ and $fib2$ could also be obtained by optimizing $fib1rec$ and $fib2rec$, respectively, using the method in this paper. Figure 4 shows the measurements.

input n	$fib(n)$	$fib1rec(n)$	$fib2rec(n)$	$fib1(n)$	$fib2(n)$	$fib3(n)$
40	26.91s	1.48	1.39	1.11	0.98	0.03
80	>200s	2.94	2.82	2.21	2.02	0.05
120	>200s	4.40	4.31	3.36	3.11	0.08
space	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$

Figure 4: Running times (in milliseconds) and space usage of Fibonacci in C

The resulting program preserves correctness in the sense that, if the original program terminates with a value, then the optimized program terminates with the same value.

8 Summary and discussion

We have described a general method by starting with the basics and gradually generalizing it. Each previous case has a restricted feature and can be considered as an optimization of a latter case. For each of the cases, we described characterization for the classes of programs, the optimizations, and the speedups. We also measure performance improvements.

We summarize the entire transformation algorithm, including all the optimizations. The algorithm starts with transforming function f_0 and repeatedly transforms functions that are called in the resulting program until all those called are transformed. To transform a function f, it performs the following three steps (the parenthesis after each step includes the section that describes the details).

Step I. Analyze f to identify base cases and recursive cases (Sec. 2). Note this can be done easily by analyzing f only because f is properly defined. If f has only base cases, then f needs not be transformed; if f has only recursive cases, then we conclude that f does not terminate and do not transform it either. In both cases, we are done with f. Otherwise, continue with Step II.

Step II. For each recursive case of f, we do the following two substeps. First, identify an input increment (Sec. 7). Second, derive an incremental version (Sec. 7). Continue with Step III.

Step III. Form iterative program. There are two cases. Case 1, f has one recursive case and the decrement operation has an inverse. If f has one base case input, then initialize using it and iterate based on the increment operation (Sec. 3). Otherwise, f has more than one base case input, then use an initial loop to decrement the input before initialize and iterate using the increment operation (Sec. 4). Case 2, f has more than one recursive case or an decrement operation has no inverse. If f is not on recursive data structures, then form an iterative program that uses the stack explicitly if it is more efficient than using stack frames during the execution (Sec. 5). Otherwise, f is on recursive date structures, then form an iterative program that uses the stack explicitly

(Sec. 5) and then optimize using pointer reversal and, if possible, eliminate backtracking and heap allocation (Sec. 6).

Continuing Step III, for both Cases 1 and 2, after transformation to iterations, two additional optimizations can be performed. First, if additional information is cached in incremental computation, then perform additional optimizations (Sec. 7). Second, if f is already tail recursive, then the incremental version will be an identity function, so eliminate the stack and the incremental computation. This finishes the transformation from recursion to iteration.

The optimizations succeed in most cases. The only failure case is when (i) f has more than one recursive case or an decrement operation has no inverse, (ii) no associativity that guarantees speedups can be exploited, (iii) f is not on recursive data structures, and (iv) allocating and deallocating a node in a singly linked list is more expensive than allocating and deallocating stack frames during the execution. We have applied the algorithm to all examples we found in the literature and it succeed in transforming all of them into iterations. Note that mutual recursions are handled uniformly; for an example, see the matrix-chain multiplication problem [34]

These optimizations guarantee that if the original program terminates with a value, then the optimized program terminates with the same value. When no elimination of useless computations is performed, non-termination is preserved as well, as seen in Sections 4, 5, and 6. These optimizations eliminate the use of stack frames and, for most cases, the use of stack space completely so the optimized program takes only constant additional space and runs significantly faster.

The entire transformation can be performed fully automatically. Step I needs a simple static analysis of a single function. Step II needs to identify input increments and derive incremental versions. Identifying input increments is straightforward for linear recursion; for non-linear recursion, the algorithm we use [34] is fully automatable and is effective on all examples we know, and furthermore, it enabled us to derive better programs than by previous methods. For example, for the Hanoi tower problem, we obtain a program that is half the size and uses half of the variables as that derived by Pettorossi and Proietti [44]. Deriving incremental versions is fully automatic modulo equality reasoning and timing analysis, but limiting both to use fully automatable techniques, we are able to incrementalize all examples we know, which include all dynamic programming examples in [2, 46, 13]. Step III consists of all transformations described in this paper, all of which are straightforward rewrites based on simple syntax analysis.

Transforming recursion to iteration is an extremely rich subject. This paper simply followed the observation that incremental computation gives rise to iterative computation, essentially for free, even if starting at non-linear recursions, and summarizes major optimizations that can be performed when forming the iterative computation. More cases and optimizations are also being formalized, notably recursion on tree-structured data, but the general idea of forming iteration using incremental computation, as well as optimizing stack-fashioned data structure traversal using pointer reversal, remains the same. Another form of iteration, as discussed below in related work, is to explore the use of arrays, rather than linked lists, for the stack data structures. Based on our previous implementation of incrementalization [31], a prototype implementation of the transformations presented here is under way.

9 Related work

Transforming recursion into iteration, often called recursive removal, as well as general relationships between recursion and iteration, have been studied extensively from theory to practice. We compare with major related works here; additional references can be found following each work.

Program schematology [42, 17, 49, 19] studies program behaviors and equivalences based on schemas with uninterpreted function symbols. Special program schemas are identified, and equivalences or inequivalences between them are proved. For example, it is shown that general recursive scheme is more powerful than while scheme, and a linear recursive scheme is equivalent to a while scheme [42]. The equivalence results may be used for program transformation, but only for the matched schemas, and the results are not specially aimed at performance improvements. For example, Paterson and Hewitt [42] show that any $O(n)$ time linear recursion can be realized in a $O(n^2)$ time while program using two memory cells. Our approach is to design a general method of program optimization by exploiting the semantics of each language construct. Thus it is fundamentally different from schema-based approach, where function symbols are uninterpreted, even though it can produce the same optimized forms when applied to given program schemas. The method is based on incrementalization, with a general principle aimed at improving the performance of repeated computations, following which important optimizations fall out. Furthermore, we explicitly use time-and-space analyses, guaranteeing performance improvement as well as correctness.

Burstall and Darlington [10] first studied transforming recursive functions, including recursion removal, using a set of transformation rules and certain strategies, notably unfold, fold, and eureka definitions. The generality and flexibility in using these rules allow many programs to be derived in one way or another but also cause individual derivations to be ad hoc, rather than systematic. Their implementation of recursion removal [14] is completely based on a set of schemas. Others have tried to make these transformations more systematic by exploiting certain principles; many of these are summarized by Partsch [41]. In particular, Wegbreit [53] studied program analysis and goal-directed transformations; Wand [50] studied the use of continuation in transforming a number of examples; Scherlis used internal specialization [47]; Cohen [12] explicitly addressed classes of redundant recursive calls; Pettorossi [43, 44] exploited tupling. These methods are more systematic but are still either not automatable or not powerful enough to derive many examples.

Backus proposed FP [5, 6] where algebraic laws can be used in forming theorems concerning transforming certain program schemas, including recursion removal. Kieburtz and Shultis [29] followed this approach and proved more general theorems, thus allowing the transformation of a bigger class of program schemas. Bauer and Wossner [7] discussed an extensive set of linear functions, even though not formalized in FP, that can be turned into iterative forms. Harrison and Khoshnevisan again followed the FP approach, proving theorems for transforming a certain class of non-linear functions into linear forms [20] and transforming linear recursion into loops [21]. FP-based approach could be easier for algebraic reasoning but at the same time is difficult to be applied to conventional programs. The resulting schema-based transformations are difficult to implement, requiring at least second-order patter matching [25], as well as ability of proving preconditions of the theorems.

Several other works involve transforming recursion to iteration, exploring special properties or being in the context of other studies. Associativity allows a kind of reversal of the order of computation and is used by many in transforming recursion to iteration [4, 41, 9], but none addresses possible slowdown, as we found for the factorial function. We consider it as a separate optimization, and a simple analysis is used to guarantee performance improvement. Waters [52] studied transforming series expressions into loops in the context of their program synthesis project. Its underlying idea is the same as incrementalization, but series expressions consist of operations on aggregates, not through recursion, so it is easier to use rules for each aggregate operation. Harrison [23] studied parallelization of Scheme programs in his dissertation work by compiling recursions to iterations using stacks that are compiled as vectors. Interprocedural analysis are used for determining stack allocation and deallocation in the presence of side effects. His techniques can handle only linear recursion. No optimizations for contracting the decrementing loop, removing the stack, eliminating heap allocation, and so on are studied. Furthermore, running times are given for a few Scheme programs with no comparison to anything else.

It is know that conversion to continuation-passing style (CPS) can turn recursions into tail forms [16]. However, a transformed program accumulates parameters in closures so it has the same asymptotic space behavior as the recursive program. Furthermore, it may be slower by a constant factor, since if closures are allocated on the heap, the overhead of garbage collection needs to be added. When callcc is not allowed, closures created by CPS transformation can be allocated on the stack, and this overhead may be eliminated; even so, there is still the overhead of closure allocation compared with simple iteration. Filinski also formulated the fact that in Scheme-like languages with first-class continuation, recursion can be characterized as a particular pattern of iteration [15]. Again, no optimizations are achieved by such transformation. It merely indicates that higher-order-ness gives rise to more expressiveness, which is precisely characterized and compared with one another by Jones [26]. Transforming recursion into iteration and comparing them at higher-orders have also been studied by Kfoury [28] in the framework of program schematology.

Efficiency of iterative programs very much depends on the efficient use of low-level data structures, such as linked list. Despite of various works on data structure selection in program refinement and program synthesis [39, 41, 48, 8], few works in transforming recursion to iteration address optimizations that arise naturally in this setting. As an example, Darlington and Burstall [14] briefly discussed reusing discarded cells. To our knowledge, no previous work achieves pointer reversal by correctness-preserving transformations as we do. Another possible optimization when forming iterative programs is to use arrays instead of linked lists. Paige [39] has studied when linked list representation can be implemented efficiently using arrays, together with automatic data structure selection in general; this is for efficient implementation of sets and maps when compiling very-high-level language SETL [40, 11]. Odersky [38] recently proposed programming with functional variables; their efficient implementation will depend heavily the usage of efficient data structures.

We have previously studied general methods for incrementalization that exploits the return value [37], the intermediate results [36], and auxiliary information [35] in a previous computation. Incrementalization is used here but is simplified: a standalone incremental version needs to compute the value for base cases, while an incremental version as used in the loop body only needs to consider the case where a previous value is used, since the base cases can be separated out in the resulting program. Incrementalization has been used for optimizing recursions, improving exponential-time programs to polynomial-time programs [34], but the optimized programs were still recursive, not iterative. This work gives rise to further drastic performance improvement in time, but more importantly in space. As a final note, loops can be further improved using incrementalization [32, 33].

Acknowledgment. The authors are grateful to the following people. Anil Nerode pointed out that incrementalization simply turns recursion into iteration several years back. Allen Brown first brought up the work on program schematology. Alberto Pettorossi first provided reference [42]. Steve Johnson pushed to see how to derive tail recursion from general recursion. Andrew Appel pointed out reference [23]. Colin Runciman suggested a reference that led to references [20] and [21]. Neil Jones explained in details his new complexity results regarding recursion, tail recursion, and higher-order types [26] and provided good insights into several related works.

References

[1] H. Abelson, R. K. Dybvig, et al. Revised report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, Aug. 1998.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1974.

[3] A. W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.

[4] J. Arsac and Y. Kodratoff. Some techniques for recursion removal from recursive functions. *ACM Trans. Program. Lang. Syst.*, 4(2):295–322, Apr. 1982.

[5] J. Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, Aug. 1978.

[6] J. Backus. From function level semantics to program transformation and optimization. In H. Ehrig et al., editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, volume 185 of *Lecture Notes in Computer Science*, pages 60–91. Springer-Verlag, Berlin, Mar. 1985.

[7] F. L. Bauer and H. Wössner. *Algorithmic Language and Program Development*. Springer-Verlag, Berlin, 1982.

[8] L. Blaine and A. Goldberg. DTRE—A semi-automatic transformation system. In B. Möller, editor, *Constructing Programs from Specifications*, pages 165–203. North-Holland, Amsterdam, 1991.

[9] E. A. Boiten. Improving recursive functions by inverting the order of evaluation. *Sci. Comput. Program.*, 18(2):139–179, Apr. 1992.

[10] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, Jan. 1977.

[11] J. Cai and R. Paige. Program derivation by fixed point computation. *Sci. Comput. Program.*, 11:197–261, Sept. 1988/89.

[12] N. H. Cohen. Eliminating redundant recursive calls. *ACM Trans. Program. Lang. Syst.*, 5(3):265–299, July 1983.

[13] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press/McGraw-Hill, 1990.

[14] J. Darlington and R. Burstall. A system which automatically improves programs. *Acta Informatica*, 6(1):41–60, 1976.

[15] A. Filinski. Recursion from iteration. *Lisp and Symbolic Computation*, 7(1):11–38, Jan. 1994.

[16] D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages*. MIT Press and McGraw-Hill, 1992.

[17] S. J. Garland and D. C. Luckham. Program schemes, recursion schemes, and formal languages. *J. Computer System Sciences*, 7:119–160, 1973. Also appeared as technical report UCLA-ENG-7154, June 1971.

[18] J. Golsing, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Reading, Mass., 1997.

[19] S. A. Greibach. *Theory of Program Structures: Schemes, Semantics, Verification*, volume 36 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1975.

[20] P. G. Harrison. Linearisation: An optimization for nonlinear functional programs. *Sci. Comput. Program.*, 10:281–318, 1988.

[21] P. G. Harrison and H. Khoshnevisan. A new approach to recursion removal. *Theoret. Comput. Sci.*, 93(1):91–113, 1992.

[22] P. G. Harrison and H. Khoshnevisan. On the synthesis of function inverses. *Acta Informatica*, 29(3):211–239, 1992.

[23] W. L. Harrison, III. The interprocedural analysis and automatic parallellization of scheme programs. *Lisp and Symbolic Computation*, 2(3/4):179–396, Oct. 1989.

[24] P. Hudak. A semantic model of reference counting and its abstraction. In *Conference Record of the 1986 ACM Symposium on LISP and Functional Programming*, pages 351–363. ACM, New York, Aug. 1986.

[25] G. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11(1):31–55, 1978.

[26] N. D. Jones. The expressive power of higher-order types or, life without CONS. *Journal of Functional Programming*. To appear.

[27] S. B. Jones and D. Le Métayer. Compile-time garbage collection by sharing analysis. In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture*, pages 54–74. ACM, New York, Sept. 1989.

[28] A. J. Kfoury. Recursion versus iteration at higher-orders. In *Foundations of Software Technology and Theoretical Computer Science*, volume 1346 of *Lecture Notes in Computer Science*, pages 57–? Springer-Verlag, Berlin, Dec. 1997.

[29] R. B. Kieburtz and J. Shultis. Transformations of fp program schemes. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*. ACM, New York, 1981.

[30] N. Klarlund. *MONA Version 1.3 User Manual*, Oct. 1998.

[31] Y. A. Liu. CACHET: An interactive, incremental-attribution-based program transformation system for deriving incremental programs. In *Proceedings of the 10th Knowledge-Based Software Engineering Conference*, pages 19–26. IEEE CS Press, Los Alamitos, Calif., Nov. 1995.

[32] Y. A. Liu. Principled strength reduction. In *Proceedings of the IFIP TC2 Working Conference on Algorithmic Languages and Calculi*. Chapman & Hall, London, U.K., Feb. 1997.

[33] Y. A. Liu and S. D. Stoller. Loop optimization for aggregate array computations. In *Proceedings of the IEEE 1998 International Conference on Computer Languages*, pages 262–271. IEEE CS Press, Los Alamitos, Calif., May 1998.

[34] Y. A. Liu and S. D. Stoller. Dynamic programming via static incrementalization. In *Proceedings of the 8th European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 288–305. Springer-Verlag, Berlin, Mar. 1999.

[35] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Discovering auxiliary information for incremental computation. In *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 157–170. ACM, New York, Jan. 1996.

[36] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Static caching for incremental computation. *ACM Trans. Program. Lang. Syst.*, 20(3):546–585, May 1998.

[37] Y. A. Liu and T. Teitelbaum. Systematic derivation of incremental programs. *Sci. Comput. Program.*, 24(1):1–39, Feb. 1995.

[38] M. Odersky. Programming with variable functions. In *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming*, pages 105–116. ACM, New York, Sept. 1998.

[39] R. Paige. Real-time simulation of a set machine on a RAM. In *Computing and Information, Vol. II*, pages 69–73. Canadian Scholars Press, 1989. Proceedings of ICCI '89: The International Conference on Computing and Information, Toronto, Canada, May 23-27, 1989.

[40] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst.*, 4(3):402–454, July 1982.

[41] H. A. Partsch. *Specification and Transformation of Programs—A Formal Approach to Software Development.* Springer-Verlag, Berlin, 1990.

[42] M. S. Paterson and C. E. Hewitt. Comparative schematology. In *Project MAC Conference on Concurrent Systems and Parallel Computation*, pages 119–127. ACM, New York, 1970.

[43] A. Pettorossi. A powerful strategy for deriving efficient programs by transformation. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming.* ACM, New York, Aug. 1984.

[44] A. Pettorossi and M. Proietti. Rules and strategies for transforming functional and logic programs. *ACM Comput. Surv.*, 28(2):360–414, June 1996.

[45] W. Pugh. The Omega Test: A fast and practical integer programming algorithm for dependence analysis. *Commun. ACM*, 31(8), Aug. 1992.

[46] P. W. Purdom and C. A. Brown. *The Analysis of Algorithms.* Holt, Rinehart and Winston, 1985.

[47] W. L. Scherlis. Program improvement by internal specialization. In *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages*, pages 41–49. ACM, New York, Jan. 1981.

[48] D. R. Smith. KIDS: A semiautomatic program development system. *IEEE Trans. Softw. Eng.*, 16(9):1024–1043, Sept. 1990.

[49] S. A. Walker and H. R. Strong. Characterizations of flowchartable recursions. *Journal of Computer and System Sciences*, 7:404–447, 1973.

[50] M. Wand. Continuation-based program transformation strategies. *J. ACM*, 27(1):164–180, Jan. 1980.

[51] M. Wand and W. D. Clinger. Set constraints for destructive array update optimization. In *Proceedings of the IEEE 1998 International Conference on Computer Languages*, pages 184–193. IEEE CS Press, Los Alamitos, Calif., May 1998.

[52] R. C. Waters. Automatic transformation of series expressions into loops. *ACM Trans. Program. Lang. Syst.*, 13(1):52–98, Jan. 1991.

[53] B. Wegbreit. Goal-directed program transformation. *IEEE Trans. Softw. Eng.*, SE-2(2):69–80, June 1976.