Optimizing Ackermann's Function by Incrementalization[*]

Yanhong A. Liu Scott D. Stoller

Computer Science Department, State University of New York at Stony Brook, Stony Brook, NY 11794, USA

{liu,stoller}@cs.sunysb.edu

ABSTRACT

This paper describes a formal derivation of an optimized Ackermann's function following a general and systematic method based on incrementalization. The method identifies an appropriate input increment operation and computes the function by repeatedly performing an incremental computation at the step of the increment. This eliminates repeated subcomputations in executions that follow the straightforward recursive definition of Ackermann's function, yielding an optimized program that is drastically faster and takes extremely little space. This case study uniquely shows the power and limitation of the incrementalization method, as well as both the iterative and recursive nature of computation underlying the optimized Ackermann's function.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*control structures, recursion*; D.3.4 [**Programming Languages**]: Processors—*optimization*; F.3.2 [**Logics and Meanings of Programs**]: Logics and Meanings of Programs—*partial evaluation*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program transformation*

General Terms

Algorithms, Languages, Performance

Keywords

Caching, incremental computation, incrementalization, iteration, memoization, program transformation, optimization, recursion, tabulation

[*]This work was supported in part by ONR under grants N00014-99-1-0132, N00014-99-1-0358, and N00014-01-1-0109 and by NSF under grant CCR-9711253 and CCR-9876058.

1. INTRODUCTION

General and systematic methods for transforming high-level programs into efficient implementations are important for programming languages, compilers, and programming methodologies. The power and limitation of such a method, as well the method itself, can be well shown through challenging case studies. This short paper presents such a case study, by applying a general and systematic method based on incrementalization to Ackermann's function.

Ackermann's function is defined recursively by

```
a(i,n) = if i=0 then n+1
         else if n=0 then a(i-1,1)
         else a(i-1,a(i,n-1))
```

It is the canonical example of a recursive function that is not primitive recursive; it grows faster than any polynomial, exponential, multiple exponential, or primitive recursive functions. A function that is roughly inverse of Ackermann's function is used for complexity analysis of algorithms, such as disjoint set union-find [8], which has applications such as program analysis [10, 14, 28]. Grossman and Zeitman [9] give a historical account of Ackermann's function.

Direct evaluation of the recursive definition requires $O(\texttt{a(i,n)})$ stack space and has much worse running time due to repeated computation of the same function calls. For example, notice that `a(i,n)` needs `a(i,n-1)`, which needs `a(i,n-2)`, and so on down to `a(i,0)`; each of the `a(i,j)` for $1 \leq \texttt{j} \leq \texttt{n}$ gets the value of `a(i-1,a(i,j-1))`, but each `a(i-1,a(i,j-1))` also needs `a(i-1,a(i,j-2))`, so each `a(i-1,a(i,j-2))` is computed repeatedly in all of `a(i,j-1)`, `a(i,j)`, ... `a(i,n)`. An optimized Ackermann's function can compute each function call once, take only $O(\texttt{i})$ space, and run drastically faster.

We derive such an optimized Ackermann's function by applying a general and systematic method based on incrementalization [15, 20]. The idea is to identify an appropriate input increment operation and compute the function by repeatedly performing an incremental computation at the step of the increment. Transforming a function into an incremental version that does an incremental computation under the input increment operation involves three steps: (1) cache all intermediate results, (2) incrementalize the function under the input increment to use the cached results, and (3) prune unused intermediate results. The resulting optimized Ackermann's function takes $O(\texttt{i})$ space and $O(\texttt{a(i,n)})$ time. The derivation based on semantics-preserving transformations also ensures the correctness of the optimized program.

Applying optimization based on incrementalization to Ackermann's function has three benefits. First, even though the transformations and analyses for incrementalization have been studied previously and used successfully on many examples [15, 19, 18, 20, 21], applying it to Ackermann's function needed a small and natural but important extension, to allow the body of an incremental function to contain repeated calls, not just one-time calls, to incremental functions. This extension has general applications for incremental computation problems in practice, where a change to a problem input needs to be handled by a sequence of incremental updates. For incrementally computing `a(i,n)` after `a(i,n-1)` has already been computed, the number of repetitions can even be as large as `a(i,n-1)`.

Second, the derivation helps explain what contributes to the optimization. Repeated incremental computation at the step of an increment is naturally an iterative computation as well as a bottom-up computation, though it can be written using tail recursion also. Besides this iterative computation, our resulting program also contains non-tail recursion, though one may use loops in place of recursion and keep the stack in a linked list or an array. Iterative in contrast to recursive forms are not what give rise to the optimization. What yields the optimization is the elimination of repeated computations, a smart form of memoization [22] and tabulation [3], as a result of incremental computation.

Third, although the optimization method based on incrementalization has been used successfully to optimize loops, arrays, recursive functions, and recursive data structures [15] and transform recursion to iteration [17], the power of the method in general remains unknown. A similar method has also been used to optimize set and fixed-point operations [4, 23, 24]. Previously, all these optimizations succeeded in deriving efficient programs, including well-known dynamic programming programs [19, 18], from more straighforward high-level programs that are exponential or high-degree polynomial. Applying it to Ackermann's function shows that it also applies to functions that are not primitive recursive. In particular, it reduces the space usage from $O(\texttt{a(i,n)})$ to $O(\texttt{i})$, while other methods using memoization and tabulation would yield programs that still need $O(\texttt{a(i,n)})$ space.

Related work

To the best of our knowledge, this is the first formal derivation of such an optimized Ackermann's function following a general and systematic method. Grossman and Zeitman [9] describe an optimized version that uses loops and arrays and takes $O(\texttt{i})$ space and $O(\texttt{i} * \texttt{a(i,n)})$ time but their optimized program was not derived and they leave the proofs to the reader; they attribute the optimization to iteration in contrast to recursion. Our optimized program is derived through semantics-preserving transformations. The derivation serves as a proof of correctness of the optimized program, even though it is not a formal proof. Our optimized program uses both recursion and iteration. Much earlier, Rice [27] gives a similar iterative procedure, without derivation, correctness proof, or complexity analysis. Berry [2] characterizes these efficient procedures for Ackermann's function as bottom-up computation [1] and demonstrates their optimality properties. Some other procedures, either similar but specialized for $\texttt{i} \leq 3$ or less efficient, are given in [29]. Jones [12] describes analysis and specialization of Ackermann's function succinctly and precisely.

Partial evaluation [11, 13] is a powerful and systematic method for program specialization; it can be used to derive a specialized version of a function for a given value of part of its input, for example, a specialized version of Ackermann's function for $\texttt{i} = 3$. It cannot derive an optimized Ackermann's function as incrementalization can, but it is used in incrementalization and is especially important for handling base cases; this is particularly needed for Ackermann's function, as will be seen in this paper. There are other program transformation methods for memoization [7, 22], tabulation [3, 7], etc.[6, 25, 26], which share similar ideas as incrementalization, but they cannot derive an optimized Ackermann's function as incrementalization can; memoizing or tabulating all function calls in computing Ackermann's function would require $O(\texttt{a(i,n)})$ space, whereas our optimized program needs only $O(\texttt{i})$ space.

Language

We use the following simple programming language. A program is a set of function definitions of the form

```
f(v1,...,vk) = e
```

and the grammar for `e` is given below. The recursive definition of Ackermann's function needs only the first four kinds of expressions, and the rest are used in the transformed programs. Tuples can have different lengths. Selectors are `1st`, `2nd`, and `3rd`, which select the first, second, and third component, respectively, of a tuple. Binding expressions are like blocks.

```
e ::= c                      constant
    | p(e,...,e)             primitive operation
    | if e then e else e     conditional expression
    | f(e,...,e)             function application
    | <e,...,e>              tuple construction
    | s(e)                   component selection
    | let b in e             binding expression
b ::= v:= e                  assignment
    | b;b                    sequencing
    | for i:= e to e do b    loop
```

2. OPTIMIZATION BY INCREMENTALIZATION

The method is based on identifying an input increment operation under which the function can be computed incrementally at the step of the increment. The goal is to reuse the results computed from one step to the next. The original function is computed by repeatedly performing this incremental computation.

First, an input increment operation is in the opposite direction of change compared to arguments of recursive calls; to allow maximum reuse, an increment that captures the minimum change is selected. For Ackermann's function `a` on argument `<i,n>`, recursive calls have arguments `<i-1,1>`, `<i,n-1>`, and `<i-1,a(i,n-1)>`. Clearly, the increment from `<i,n-1>` to `<i,n>` is the minimum, based on component-wise absolute difference. So the input increment operation takes `<i,n-1>` to `<i,n>`, i.e., it increments `n` by `1`.

Next, we derive an incremental version of the function under the input increment operation, i.e., a function that computes `a(i,n)` efficiently using the computation results about `a(i,n-1)`. Since the values of intermediate function calls computed in `a(i,n-1)`, not just the value of `a(i,n-1)`, might also be used for computing `a(i,n)`, we use a method called cache-and-prune [15, 20] that determines and maintains the appropriate intermediate results and uses them for the incremental computation. This yields a function `aUse` and an incremental function `aUse'`, such that `aUse(i,n)` returns a tree of useful intermediate results, where the value of `a(i,n)` is the leftmost child and can be retrieved using the selector `1st`, and `aUse'(i,n,rUse)` computes `aUse(i,n)` using the result `rUse` of `aUse(i,n-1)`. Functions `aUse` and `aUse'` satisfy

`1st(aUse(i,n))` = `a(i,n)`, and
if `rUse` = `aUse(i,n-1)`, then `aUse'(i,n,rUse)` = `aUse(i,n)`.

Finally, an optimized program `aOpt` is formed by initializing using `aUse` for `n = 0` and repeatedly calling the incremental function `aUse'` for `n > 0`. This can be written using either recursion

```
aOpt(i,n)
= 1st(aUseOpt(i,n))
aUseOpt(i,n)
= if n=0 then aUse(i,0)
  else let v:= aUseOpt(i,n-1)
       in aUse'(i,n,v)
```

or iteration

```
aOpt(i,n)
= let v:= aUse(i,0);
      for j:= 1 to n do
        v:= aUse'(i,j,v)
  in 1st(v)
```

The call to `aUse` will be replaced with calls to `aUse'` as well, as we will see at the end. The loop can also be rewritten equivalently using a tail recursive function. For efficiency and simplicity, we will use the `for`-loop.

The derivation of an incremental version of `a` under the input increment operation has three steps. Step 1 constructs a function `aAll`, an extended version of `a`, such that `aAll(i,n)` returns a tree of all intermediate results computed in `a(i,n)`, where the value of `a(i,n)` is the leftmost child, i.e.,

`1st(aAll(i,n))` = `a(i,n)`.

Step 2 constructs a function `aAll'`, an incremental version of `aAll` under the input increment operation, such that `aAll'(i,n,rAll)` computes `aAll(i,n)` incrementally using the result of `aAll(i,n-1)`, i.e.,

if `rAll` = `aAll(i,n-1)`, then `aAll'(i,n,rAll)` = `aAll(i,n)`.

Thus, `1st(aAll'(i,n,rAll))` = `1st(aAll(i,n))` = `a(i,n)`. Step 3 produces two functions `aUse` and `aUse'`, pruned versions of `aAll` and `aAll'`, respectively, such that `aUse(i,n)` returns only the leftmost and other intermediate results in `rAll` that are useful in computing `1st(aAll'(i,n,rAll))` where `rAll` = `aAll(i,n)`, and `aUse'(i,n,rUse)` where `rUse` = `aUse(i,n)` returns only those useful results as `aAll'(i,n,rAll)` does. These steps are described below and the final optimized program is given at the end.

2.1 Caching all intermediate results

This step constructs the function `aAll` that returns a tree, as nested tuples, containing the values of all intermediate function calls made in computing `a`; the leftmost child of a tree value is always the originally returned value. We obtain

```
aAll(i,n)
= if i=0 then <n+1>
  else if n=0 then aAll(i-1,1)
  else let v1:= aAll(i,n-1);
           v2:= aAll(i-1,1st(v1))
       in <1st(v2), v1, v2>
```

Note on caching structures: We use an untyped language for simplicity; alternatively, one could use different constructors for tuples of different lengths. The general method for caching all intermediate results caches the value of each function call in a separate component in the tail of the return tuple; applied to `a`, the second branch would be `let v:= aAll(i-1,1) in <1st(v),v>`. However, if the only function call in a branch is a tail call, then we may simply return its value without making a new tuple and caching this value separately. This simplification is based on the same idea as the optimization that avoids caching values that are embedded in the return value [21], but this special case is not covered by the optimization there. This simplification makes the code a little neater, but we can obtain essentially the same optimized program at the end without it. Note that we cannot simply say that we do not cache separately values of all tail calls, which include, for example, the call `a(i-1,a(i,n-1))` in the third branch, by making, for example, the third branch return `<v2,v1>` or `append(v2,v1)`.

The former is not feasible since, in general, to make the optimized program simple, we need to be able to retrieve the original value uniformly, for example, here, from the first component of the extended function, not the first of first for this particular branch. The latter is not feasible since, in general, we need a tree-structured data, not a flattened list, to be able to access all cached components efficiently.

2.2 Incrementalizing under the increment to use cached results

This step constructs incremental function `aAll'(i,n,rAll)` that computes `aAll(i,n)` incrementally using the cached result `rAll` of `aAll(i,n-1)`. This starts with introducing function `aAll'`.

```
aAll'(i,n,rAll) = aAll(i,n),
        where rAll = aAll(i,n-1).          (1)
```

The idea is to replace subcomputations in `aAll(i,n)` whose values can be retrieved from the cached result `rAll` of `aAll(i,n-1)`, thus the resulting function `aAll'` uses `rAll` in addition to `i` and `n`. The method exploits data structures and control structures—to use components of structured data, `rAll`, and to use them under appropriate conditions that appear in `aAll(i,n)` and `aAll(i,n-1)`—and replaces function calls made in `aAll(i,n)` with retrievals from `rAll` or with calls to incremental functions.

First, `aAll(i,n)` is expanded according to the definition of `aAll`. To allow all conditional tests `i = 0` and `n = 0` in `aAll(i,n)` and `i = 0` and `n-1 = 0` in `aAll(i,n-1)` to be exploited, the third branch of `aAll(i,n)` is duplicated with the additional test `n-1 = 0` that appears in `aAll(i,n-1)`.

```
aAll'(i,n,rAll)
= aAll(i,n)
= if i=0 then <n+1>
  else if n=0 then aAll(i-1,1)
  else if n-1=0 then
       let v1:= aAll(i,n-1);
           v2:= aAll(i-1,1st(v1))
       in <1st(v2), v1, v2>
  else let v1:= aAll(i,n-1);
           v2:= aAll(i-1,1st(v1))
       in <1st(v2), v1, v2>
```

Then, replace function calls to `aAll` in each of the branches, under the respective condition of the branch, with retrievals from `rAll` or with recursive calls to `aAll'`. The first branch simply returns `<n+1>` and is left unchanged.

In the second branch, the condition is the conjunction of $i \neq 0$ and $n = 0$. Function call `aAll(i-1,1)` cannot be computed using `rAll` or any component of `rAll` but, by (1), equals recursive call `aAll'(i-1,1,aAll(i-1,0))`. Recall that this branch is computing `aAll(i,0)`, and now it needs `aAll(i-1,0)`. Notice that `aAll(0,0) = <1>` by definition of `aAll`. Thus, we have

```
aAll(i,0) = if i=0 then <1>
            else aAll'(i-1,1,aAll(i-1,0))
```

The result of this branch is as follows, using a direct recursive function

```
aOAll(i)
where aOAll(i)
      = if i=0 then <1>
        else aAll'(i-1,1,aOAll(i-1))
```

or an iterative loop

```
let v:= <1>;
    for k:= 1 to i do
      v:= aAll'(k-1,1,v)
in v
```

The loop can also be rewritten equivalently using a tail recursive function. For efficiency and simplicity, we will use the `for`-loop for the rest of the paper.

Note on specialization: This introduction of recursion or iteration is new in the sense that no previous examples of incrementalization use it. Indeed, the transformation for this branch is a kind of specialization [11, 13] during incrementalization, where `aOAll` may be viewed as an incremental Ackermann's function specialized with respect to the second argument being `0`. As we can see with the optimized program at the end, this branch of the incremental function is actually never executed, i.e., it is dead, and can be eliminated, but the same computation is needed for initialization by the optimized program that calls the incremental function.

In the third branch, the condition is the conjunction of $i \neq 0$, $n \neq 0$, and $n-1 = 0$. First, exploit the result `rAll = aAll(i,n-1)` under this condition. Since `n-1 = 0`, by definition of `aAll` we have `aAll(i,n-1) = aAll(i-1,1)`. Therefore,

```
rAll = aAll(i,n-1) = aAll(i-1,1).          (2)
```

Then, consider the two function calls bound to `v1` and `v2` respectively. The first call `aAll(i,n-1)` is replaced with `rAll`, since `rAll = aAll(i,n-1)`. Note that, under `n-1 = 0`, `aAll(i,n-1)` also equals `aAll(i,0)` and thus `aOAll(i)`, but this is more expensive than `rAll`. The second call `aAll(i-1, 1st(v1))` equals `aAll(i-1,a(i,n-1))`. By (2), `aAll(i,n-1)` computed `aAll(i-1,1)` but not `aAll(i-1,a(i,n-1))`. Since `a(i,n-1) = a(i,0)` and `a(i,0) > 1` under `n-1 = 0` and $i \neq 0$, which can be shown by a simple induction, we can start with `aAll(i-1,1)` and incrementally compute `aAll(i-1,2)`, `aAll(i-1,3)`, ..., `aAll(i-1,a(i,n-1))` using `aAll'`, as follows.

```
let v2:= aAll(i-1,1);
    for k:= 2 to a(i,n-1) do
      v2:= aAll'(i-1,k,v2)
in v2
where aAll(i-1,1) = rAll by (2)
      a(i,n-1) = 1st(rAll) by (1)
```

In the argument of the call to `aAll'`, `v2 = aAll(i-1,k-1)`, and as the result of the call to `aAll'`, `v2 = aAll(i-1,k)`.

Note on repeated updates: This introduction of iteration is new, in the sense that no previous examples of incrementalization used it. It is also different from what happened in the second branch, and we found it to be a natural extension to the incrementalization method. Previously, if an incremental version `f'(n,r)` is introduced to compute `f(n)` using the value `r = f(n-1)`, then at a call `f(i)` where `r1 = f(i-1)` holds we can replace `f(i)` with `f'(i,r1)`, but here, only `r2 = f(j)` for some `j < i` holds, so we can start with `f(j)` and call `f'` repeatedly to compute each of `f(j+1)`, `f(j+2)`, ..., `f(i)` incrementally. This extension has general applications for incremental computation problems in practice, where a change to a problem input is often larger than minimum. Such a change can be handled by a sequence of incremental updates. This is the best approach for many applications, for example, incremental graph reachability [4] and incremental multi-pattern matching in trees [5]. This method applies if the sequence of updates together are cheaper than computing the new output from scratch. For optimizing Ackermann's function, this is true because the incremental version reuses values of all function calls computed previously while the original program recomputes them. In general, program cost analysis [16] is needed. It is interesting to note that, even though this extension is natural and important, it is not needed to optimize all the well-known dynamic programming problems [19, 18].

In the fourth branch, the condition is the conjunction of `i` $\neq$ `0`, `n` $\neq$ `0`, and `n-1` $\neq$ `0`. Again, first, exploit `rAll = aAll(i,n-1)`. Since `i` $\neq$ `0` and `n-1` $\neq$ `0`, by definition of `aAll` we have

```
aAll(i,n-1) = let v1:= aAll(i,n-2);
                  v2:= aAll(i-1,1st(v1))
              in <1st(v2),v1,v2>
```

Therefore,

```
aAll(i,n-2) = v1 = 2nd(rAll)  and
aAll(i-1,a(i,n-2)) = v2 = 3rd(rAll).          (3)
```

Consider the two function calls in the fourth branch, bound to `v1` and `v2` respectively. As in the third branch, the first call `aAll(i,n-1)` is replaced with `rAll`. The second call `aAll(i-1,1st(v1))` equals `aAll(i-1,a(i,n-1))`. By (3), `aAll(i,n-1)` computed `aAll(i-1,a(i,n-2))` but not `aAll(i-1,a(i,n-1))`. Similar to the third branch, since `a(i,n-1) > a(i,n-2)`, we can start with `aAll(i-1,a(i,n-2))` and incrementally compute `aAll(i-1,a(i,n-2)+1)`, `aAll(i-1,a(i,n-2)+2)`, ..., `aAll(i-1,a(i,n-1))` using `aAll'`, as follows.

```
let v2:= aAll(i-1,a(i,n-2));
    for k:= a(i,n-2)+1 to a(i,n-1) do
      v2:= aAll'(i-1,k,v2)
in v2
where aAll(i-1,a(i,n-2)) = 3rd(rAll) by (3)
      a(i,n-2) = 1st(2nd(rAll)) by (3)
      a(i,n-1) = 1st(rAll) by (1)
```

Putting the results of all four branches together, we obtain the following incremental function.

```
aAll'(i, n, rAll)
= if i=0 then <n+1>
  else if n=0 then
       let v:= <1>;
           for k:= 1 to i do
             v:= aAll'(k-1,1,v)
       in v
  else if n-1=0 then
       let v1:= rAll;
           v2:= rAll;
           for k:= 2 to 1st(rAll) do
             v2:= aAll'(i-1,k,v2)
       in <1st(v2), v1, v2>
  else let v1:= rAll;
           v2:= 3rd(rAll);
           for k:=1st(2nd(rAll))+1 to 1st(rAll) do
             v2:= aAll'(i-1,k,v2)
       in <1st(v2), v1, v2>
```

2.3 Pruning unused intermediate results

This step prunes `aAll` and `aAll'` to obtain `aUse` and `aUse'` that together cache, use, and maintain only intermediate results needed for incrementally computing the first component of `aAll`. This is based on a backward dependence analysis of `aAll'`, starting at the first component of its return value. We find that arguments `i`, `n`, and the first and third components of `rAll` are needed but, in the second component of `rAll`, only the first subcomponent is needed. Eliminating other subcomponents of the second component, we obtain the following pruned functions.

```
aUse(i,n)
= if i=0 then <n+1>
  else if n=0 then aUse(i-1,1)
  else let v11:= a(i,n-1);
           v2:= aUse(i-1,v11)
       in <1st(v2), v11, v2>

aUse'(i, n, rUse)
= if i=0 then <n+1>
  else if n=0 then
       let v:= <1>;
           for k:= 1 to i do
             v:= aUse'(k-1,1,v)
       in v
  else if n-1=0 then
       let v11:= 1st(rUse);
           v2:= rUse;
           for k:= 2 to 1st(rUse) do
             v2:= aUse'(i-1,k,v2)
       in <1st(v2), v11, v2>
```

```
        else let v11:= 1st(rUse);
                 v2:= 3rd(rUse);
                 for k:= 2nd(rUse)+1 to 1st(rUse) do
                   v2:= aUse'(i-1,k,v2)
             in <1st(v2), v11, v2>
```

Finally, an optimized program is formed by initializing using `aUse` for `n = 0` and repeatedly calling the incremental version `aUse'` for `n > 0`. Notice that, if `n = 0`, `aUse(i,n) = aUse(i,0)`, which is computed in the second branch of `aUse'`. Therefore, we can use the second branch of `aUse'` for the initialization. We obtain the following optimized program `aOpt`, where `aUse'` is as defined above.

```
    aOpt(i,n)
    = let v:= <1>;
          for k:= 1 to i do
            v:= aUse'(k-1,1,v);
          for j:= 1 to n do
            v:= aUse'(i,j,v)
      in 1st(v)
```

Note on the final program: When `aUse'` is called, the second argument is never `0`, so the second branch of `aUse'` is dead and can be eliminated. Although `aUse` is not used in the final program, it plays an important role in the transformation.

3. ANALYSIS AND EXPERIMENTS

The final optimized program `aOpt` takes $O(\texttt{i})$ live space, for the following reasons. First, `aOpt` calls `aUse'` iteratively, and `aUse'` calls itself recursively but its first argument is smaller than `i` and decreases strictly. Second, each call to `aUse'` requires constant space, because it allocates a few local variables and creates one triple, whose first two components are numbers and whose third component is the result of a recursive call.

Running time of the optimized program is $O(\texttt{a(i,n)})$. First, the only operations Ackermann's function performs are recursive calls and adding `1`, together from `1` to `a(i,n)`, although the straightforward function computes many repeated recursive calls. The optimized program computes `a(i,n)` incrementally based on what is computed by `a(i,n-1)`, and thus there are no repeated computations; it only performs new recursive calls and, for the base cases of `i = 0`, adds by `1`. Second, each recursive call evaluates to a distinct number except for pairs like `a(i,n)` and `a(i-1,a(i,n-1))`, but such pairs become rarer at an increasing rate as `i` increases, so the total number of recursive calls is $O(\texttt{a(i,n)})$.

Grossman and Zeitman [9] say that the time complexity has an additional `i` factor, which is unnecessary. They also say that their program is inherently iterative, but it basically uses two arrays in place of the stack. They also leave the proof of correctness to the reader, but the proof is non-trivial.

We implemented the straightforward Ackermann's function and the optimized Ackermann's function in C. Tuples are implemented as records in the optimized function. The program was compiled with gcc 2.96 and uses the Boehm-Demers conservative garbage collector 5.3 to collect dynamically allocated records. We measured the running times on a 733MHz Pentium III with 256KB cache and 128MB memory running Red Hat Linux 7.0. The optimized program computes drastically faster than the straightforward program and takes extremely little space. For example, the straightforward program took about 6 minutes to compute `a(4,1)`, which equals `a(3,13)` and equals 65533, and the memory usage increased steadily to about 2.4MB from a jump start of about .5MB, but the optimized program took only 0.04 seconds. Also, the optimized program computed `a(3,20)`, which equals `8388605`, in 5.05 seconds and computed `a(3,30)`, which equals `8589934589`, in about 87 minutes, and the memory usage stayed constantly at about .5MB. Note that 64-bit long integers are used, because computing `a(3,30)` would cause overflows using 32-bit integers. Grossman and Zeitman [9] report that their optimized program, written in Pascal and using only loops and statically allocated arrays, computed `a(4,1)` in about 12 minutes. Assuming the measurement was taken in 1986 when the paper was submitted, then based on Moore's law that CPU speed doubles every 18 months, their algorithm would be 645 ($= 2^{14/1.5}$) times faster in 2000, the vintage of the machine used for our measurements, and thus take about 1.12 (= 12*60/645) seconds. Our optimized program is about 30 times faster than this, and it can be made even faster by using loops and statically allocated arrays instead of dynamically allocated and deallocated stack (for recursion) and heap (for records).

Acknowledgment

Anil Nerode pointed out in 1995 that our incrementalization-based method could derive an iterative version of Ackermann's function. We deeply appreciate his encouragement for our work on incrementalization all along. Work on partial evaluation as described by Neil Jones has been an important source of inspiration. We deeply appreciate his insightful comments on our derivation and his encouragement for our work, which contributed partly to our writing of this paper. We also thank Alan Mycroft for his interest in incrementalization and his specific questions about Ackermann's function that eventually led us to optimize it using incrementalization.

4. REFERENCES

[1] G. Berry. Bottom-up computation of recursive programs. *RAIRO Informatique Thorique/Theoretical Computer Science*, 10(1):47–82, Mar. 1976.

[2] G. Berry. Calculs ascendants du programme d'Ackermann: Analyse du programme de J. Arsac. *RAIRO Informatique Thorique/Theoretical Computer Science*, 11(2):113–126, 1976.

[3] R. S. Bird. Tabulation techniques for recursive

programs. *ACM Comput. Surv.*, 12(4):403–417, Dec. 1980.

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

[5] J. Cai, R. Paige, and R. Tarjan. More efficient bottom-up multi-pattern matching in trees. *Theoret. Comput. Sci.*, 106(1):21–60, Nov. 1992.

[6] W.-N. Chin. Towards an automated tupling strategy. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 119–132. ACM, New York, 1993.

[7] W.-N. Chin and M. Hagiya. A bounds inference method for vector-based memoization. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 176–187. ACM, New York, 1997.

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

[9] J. W. Grossman and R. S. Zeitman. An inherently iterative computation of Ackermann's function. *Theoretical Computer Science*, 57(2-3):327–330, May 1988.

[10] F. Henglein. Efficient type inference for higher-order binding-time analysis. In *Proceedings of the 5th International Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 448–472. Springer-Verlag, Berlin, 1991.

[11] N. D. Jones. An introduction to partial evaluation. *ACM Comput. Surv.*, 28(3):480–503, 1996.

[12] N. D. Jones. Notes on Ackermann's function and comments on derivation. Email communications, Oct. 2000.

[13] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.

[14] T. Lengauer and R. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, July 1979.

[15] Y. A. Liu. Efficiency by incrementalization: An introduction. *Higher-Order and Symbolic Computation*, 13(4):289–313, Dec. 2000.

[16] Y. A. Liu and G. Gómez. Automatic accurate cost-bound analysis for high-level languages. *IEEE Transactions on Computers*, 50(12):1295–1309, Dec. 2001. An earlier version appeared in *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, 1998.

[17] Y. A. Liu and S. D. Stoller. From recursion to iteration: what are the optimizations? In *Proceedings of the ACM SIGPLAN 2000 Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 73–82. ACM, New York, 2000.

[18] Y. A. Liu and S. D. Stoller. Program optimization using indexed and recursive data structures. In *Proceedings of the ACM SIGPLAN 2002 Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 108–118. ACM, New York, 2002.

[19] Y. A. Liu and S. D. Stoller. Dynamic programming via static incrementalization. *Higher-Order and Symbolic Computation*, 16(1-2):37–62, Mar.-June 2003. An earlier version appeared in *Proceedings of the 8th European Symposium on Programming*, 1999.

[20] 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. An earlier version appeared in *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 1995.

[21] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Strengthening invariants for efficient computation. *Sci. Comput. Program.*, 41(2):139–172, Oct. 2001. An earlier version appeared in *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, 1996.

[22] D. Michie. "memo" functions and machine learning. *Nature*, 218:19–22, Apr. 1968.

[23] 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.

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

[25] 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, 1984.

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

[27] H. G. Rice. Recursion and iteration. *Commun. ACM*, 8(2):114–115, Feb. 1965.

[28] B. Steensgaard. Points-to analysis in almost linear time. In *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 32–41. ACM, New York, 1996.

[29] M. P. Ward. Iterative procedures for computing Ackerman's function. Technical Report 9, Department of Computer Science, University of Durham, Durham, U.K., July 1993.