

# Program optimization using indexed and recursive data structures

Yanhong A. Liu

Scott D. Stoller

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

liu@cs.sunysb.edu

stoller@cs.sunysb.edu

## ABSTRACT

This paper describes a systematic method for optimizing recursive functions using both indexed and recursive data structures. The method is based on two critical ideas: first, determining a minimal input increment operation so as to compute a function on repeatedly incremented input; second, determining appropriate additional values to maintain in appropriate data structures, based on what values are needed in computation on an incremented input and how these values can be established and accessed. Once these two are determined, the method extends the original program to return the additional values, derives an incremental version of the extended program, and forms an optimized program that repeatedly calls the incremental program. The method can derive all dynamic programming algorithms found in standard algorithm textbooks. There are many previous methods for deriving efficient algorithms, but none is as simple, general, and systematic as ours.

## 1. INTRODUCTION

Many algorithmic problems can be described using recursive functions with little or no concern about how the functions can be computed efficiently using what data structures. Dynamic programming problems [12] are notable examples, where a straightforward recursion solves common subproblems repeatedly and takes exponential time if executed directly. A dynamic programming algorithm solves every subproblem just once, saves the result in a table, and reuses the result when the subproblem is encountered again. Such algorithms have applications in many important fields of computer science.

Important and interesting questions in general are: Can efficient algorithms and their implementation be obtained by applying transformations to the original description? Can the transformation steps be made systematic so that it is possible to provide automated support? Positive answers to these questions are important, partly because of the speedup

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PEPM '02, Jan. 14-15, 2002 Portland, OR, USA  
©2002 ACM ISBN 1-58113-455-X/02/01...\$5.00

of the resulting algorithm. Equally or more important are the assurance of correctness for the algorithm and implementation developed and the ability to develop them easily and quickly. Much previous research has answered these questions positively for subclasses of problems, e.g., [4, 7, 8, 9, 10, 11, 13, 14, 16, 19, 30, 31, 34, 37]. This paper describes a method that unifies and generalizes existing classes of problems that can be solved systematically.

Important and interesting questions in detail are: How are appropriate data structures determined? In case multiple choices exist, how can they be compared? Two fundamental classes of data structures are: indexed data structures, such as arrays, and recursive data structures, such as linked lists. In well-known graph algorithms, they correspond to representations using adjacency matrix and adjacency list, respectively. The trade offs are well-known, e.g., the former allows indexed access in constant time whereas the latter does not. Previous methods exploit many aspects of using indexed data structures or recursive data structures for optimizing recursive functions. In addition, Paige studied a systematic method that can use both linked and indexed data structures in implementing a set-based language [27]. Paige and Koenig also studied the use of auxiliary maps for efficient incremental computation of set expressions [28]. What has been lacking is a systematic method that can use both indexed and recursive data structures for optimizing recursive functions.

This paper describes such a method that is both simple and powerful. The method is based on two critical ideas; first, determining a minimal input increment operation so as to compute a function on repeatedly incremented input; second, determining appropriate additional values to maintain in appropriate data structures, based on what values are needed in computation on an incremented input and how these values can be established and accessed. Once these two are determined, the method extends the original program to return the additional values, derives an incremental version of the extended program, and forms an optimized program that repeatedly calls the incremental program.

The method can derive all dynamic programming algorithms found in standard algorithm textbooks [1, 12, 33]. We discuss in some detail the 0-1 knapsack problem, where an array is necessary for simple dynamic programming, and the binomial coefficients problem, where an array or a linked list can be used. We also discuss the single-source and all-

pairs shortest path problems and summarize more problems and experiments. We currently have a semi-automatic implementation, but we believe that a fully automated system, with a few heuristics, can derive all these examples and many more. Even though previous methods can derive efficient algorithms for many of these classical problems, no previous method is as simple, general, and systematic as ours.

The rest of this paper is organized as follows. Section 2 formulates the problem and outlines our approach. Section 3 describes our optimization method that uses indexed data structures. Section 4 describes how to use recursive data structures also. Section 5 discusses more examples and summarizes our experiments. Section 6 compares with related work and concludes.

## 2. PROBLEM FORMULATION AND APPROACH OVERVIEW

Many combinatorics and optimization problems can be solved straightforwardly using simple recursions [33, 12]. For example, consider the 0-1 knapsack problem [12]. Given  $n$  items, where the  $i$ th item is of value  $v_i$  and positive integer weight  $w_i$ , the problem is to find the maximum value for a subset of these items whose total weight does not exceed  $W$ . This can be computed as  $knap(n, W)$ , where  $knap(i, u)$  computes the maximum value for items 1 through  $i$  whose total weight does not exceed  $u$  and can be defined as, for  $i \geq 0$  and  $w_i$ 's  $> 0$ ,

$$knap(i, u) = \begin{cases} 0 & \text{if } i = 0 \text{ or } u \leq 0 \\ knap(i-1, u) & \text{if } i > 0, u > 0, \text{ and } w_i > u \\ \max(v_i + knap(i-1, u - w_i), & \text{otherwise} \\ \quad knap(i-1, u)) & \end{cases}$$

Such recursive functions can be written straightforwardly in the following first-order call-by-value functional programming language. A program is a function  $f_0$  defined by a set of mutually recursive functions of the form

$$f(v_1, \dots, v_n) = e$$

where an expression  $e$  is given by the grammar

$e ::= v$	variable
<b>for</b> $i := e_1$ <b>to</b> $e_2$ <b>do</b>	
$a[i] := e_3$	array construction
$e_1[e_2]$	array element access
$c(e_1, \dots, e_n)$	constructor application
$\tilde{c}_i(e_1)$	selector application
$c?(e_1)$	tester application
$p(e_1, \dots, e_n)$	primitive function application
<b>if</b> $e_1$ <b>then</b> $e_2$ <b>else</b> $e_3$	conditional expression
<b>let</b> $v = e_1$ <b>in</b> $e_2$	binding expression
$f(e_1, \dots, e_n)$	function application

This language allows array construction using loops, and array element access using indices. The expression **for**  $i := i_1$  **to**  $i_2$  **do**  $a[i] := v_i$  constructs an array  $a$  with elements indexed  $i_1, i_1 + 1, \dots, i_2$  and where  $a[i]$  has value  $v_i$ , and returns the array; variables  $i$  and  $a$  are bound locally, so the

name  $a$  is immaterial and serves only as a familiar syntax. The language also allows construction and access of recursive structures using constructors, selectors, and testers together with recursion. For convenience, we allow global variables, i.e., variables that are always bound to the same values, to be implicit parameters to functions. For example, a solution to the 0-1 knapsack problem can be written in our language as follows.

```
knap(i, u)
= if i = 0 ∨ u ≤ 0 then 0
  else if w[i] > u then knap(i - 1, u)
  else max(v[i] + knap(i - 1, u - w[i]), knap(i - 1, u))
```

where the arrays  $v$  and  $w$  of values and weights, respectively, are implicit parameters to  $knap$ .

Such straightforward programs may solve common subproblems repeatedly and take exponential time if executed directly. For example, suppose each item is of weight 1, then  $knap(i, u)$  calls  $knap(i-1, u-1)$  and  $knap(i-1, u)$ , where the former calls  $knap(i-2, u-2)$  and  $knap(i-2, u-1)$ , and the latter calls  $knap(i-2, u-1)$  and  $knap(i-2, u)$ , which blows up exponentially and may take  $O(2^n)$  time. We transform such programs into efficient dynamic programming algorithms that perform appropriate caching and take polynomial time in input values.

We use an asymptotic cost model for measuring time complexity. Assuming that all primitive functions take constant time, we need to consider only values of function applications as candidates for caching. Caching takes extra space, which reflects the well-known trade-off between time and space. Our primary goal is to improve the asymptotic running time of the program. Our secondary goal is to save space by caching only values useful for achieving the primary goal.

Our approach is to make computation proceed in an incremental fashion. The method has four steps. Step 1 determines how a computation should proceed, i.e., proceed at what input increment. Step 2 determines what additional values to cache based on the increment and extends the original program to return both these values and the original return value in appropriate data structures. Step 3 transforms the extended program on incremented input to use and maintain the cached values, yielding an incremental version of the extended program. Step 4 forms an optimized extended program by starting at appropriate base cases and calling the incremental extended program at each increment, and finally retrieving the original return value from the optimized extended program.

For a function  $f$  in the original program, we use  $x$  to denote its parameter, which may be a tuple of multiple parameters. We use  $next(x)$  to denote the increment operation for  $x$ . We use  $fExt$  to denote the extended function that returns additional values and the original return value, and use  $rExt$  to denote the return value of  $fExt(x)$ . We use  $fExt'$  to denote the incremental extended function that computes  $fExt(next(x))$  using the result  $rExt$  of  $fExt(x)$ , and use  $rExt'$  to denote the return value of  $fExt'(x, rExt)$ . We use  $fExtOpt$  and  $fOpt$  to denote the optimized versions of  $fExt$  and  $f$ , respectively.

### 3. SYSTEMATIC OPTIMIZATION USING INDEXED DATA STRUCTURES

Our method optimizes a given recursive function  $f$ . The method transforms the function to compute in an incremental fashion, by identifying appropriate input increments and caching, reusing, and maintaining appropriate values for the incremental computation. This can eliminate repeated sub-computations in  $f$  and yield exponential speedup. In general, this method can be applied to any function; in the worst case, the transformations are not beneficial and the function computes as before. The overall algorithm is summarized in Figure 1 and is explained in detail below.

#### 3.1 Step 1. Determine how computations should proceed

Efficient computation proceeds by processing data in an incremental fashion. The critical question is: what is the appropriate increment for input to a recursively defined function  $f$ ?

For example, the famous Fibonacci function  $fib(n)$  is defined to be 1 if  $n = 0$  or  $n = 1$ , and  $fib(n - 1) + fib(n - 2)$  otherwise. An efficient way to compute  $fib(n)$  is to compute  $fib(i)$  for  $i = 0, 1, 2, \dots, n$  at the increment of 1. Why is the increment 1 appropriate? Why not 2, or 3, or some parameter  $k$ , or -1?

For another example, the popular binomial coefficient function  $bin(n, k)$ , where  $0 \leq k \leq n$ , is defined to be 1 if  $k = 0$  or  $k = n$ , and  $bin(n - 1, k - 1) + bin(n - 1, k)$  otherwise. An efficient way to compute  $bin(n, k)$  is to compute and cache  $bin(i, k)$  for  $i = k, k + 1, \dots, n$  at the increment of 1 for the first argument and 0 for the second argument, and for each  $bin(i, k)$ , compute  $b(i, j)$  for  $j = 0, 1, \dots, k$ . Why is it appropriate to consider increment of 1 to the first argument first? Why not the second argument first, or both arguments at the same time, or increment of 2 or more?

In general, a computation may proceed incrementally in multiple ways, depending on the operations involved. There is no general method for identifying all different ways or the most appropriate ones. However, we have found an extremely simple method that can systematically determine a general class of them: simply let the increment be a minimal change from arguments of recursive calls to parameters of the defining functions. The rationale is that minimizing change corresponds to maximizing reuse, the essence of our overall approach.

Precisely, to determine how a recursive function  $f$  can be computed incrementally, first identify all possible recursive calls to  $f$ . For example, for the Fibonacci function  $fib(n)$ , recursive calls to  $fib$  are  $fib(n - 1)$  and  $fib(n - 2)$ ; for the binomial coefficient function  $bin(n, k)$ , recursive calls to  $bin$  are  $bin(n - 1, k - 1)$  and  $bin(n - 1, k)$ ; for the 0-1 knapsack problem  $knap(n, u)$ , recursive calls to  $knap$  are  $knap(i - 1, u)$ , with two occurrences, and  $knap(i - 1, u - w[i])$ . Then, for each possible recursive call, represent the arguments in terms of the parameters (including implicit parameters) of  $f$ . For all examples above, this simply yields the argument expressions themselves.

Next, among these argument expressions, identify one that corresponds to a minimal change from the parameters of  $f$ . The amount of change is measured using a partial order: a change involving fewer parameters is smaller; a difference in one parameter with smaller magnitude is smaller; other differences are incomparable. For example, for  $fib(n)$ , this is  $n - 1$ ; for  $bin(n, k)$ , this is  $\langle n - 1, k \rangle$ ; for  $knap(i, u)$ , this is  $\langle i - 1, u \rangle$ . This gives a decrement operation. Finally, taking the opposite of the decrement yields an appropriate increment operation. For example, for  $fib(n)$ , this is  $n + 1$ ; for  $bin(n, k)$ , this is  $\langle n + 1, k \rangle$ ; for  $knap(i, u)$ , this is  $\langle i + 1, u \rangle$ . If multiple minimal changes exist, all of them may be used.

How does the algorithm work for arbitrary functions? When arguments of recursive calls to  $f$  involve intermediate computations not directly using the parameters of  $f$ , the algorithm extracts expressions for all intermediate computations and simplifies the conjunction of them. In particular, when multiple function definitions are involved, i.e.,  $f$  is called recursively from other functions that are called from  $f$ , the algorithm extracts argument expressions for each intermediate call sites and equates them with the corresponding function definitions. For arbitrary functions, the simplified expression could still be too complicated for identifying minimal decrements or taking inverses of decrements, if they exist at all; however, for all actual problems we have encountered, where recursive functions describe clear solutions at a high level, this has not occurred. We currently use Omega [32] and MONA [17], which are fully automatic, to help simplify arithmetic and boolean expressions. We currently identify minimal and take inverses manually using a few facts about integer arithmetic and data construction and destruction.

This method for determining increment operations provides an answer to the questions about  $fib$  and  $bin$  posed earlier in this section. Determining input increments is theoretically hard in general, as they correspond to well-founded orderings in domain theory and steps for induction in proof theory. Yet, our method is simple and powerful. It has been used easily and successfully on all the problems we found in standard algorithm textbooks and program optimization literature; for example, it allows us to derive an iterative Ackermann's function [21]. We have not seen such a method in any textbooks or previous papers.

#### 3.2 Step 2. Determine what and how to cache selectively

First, determine additional values that need to be computed and cached for  $f$  at each incremental computation step. This is done by identifying, in the computation of  $f$  on the incremented input, all possible subcomputations, i.e., function calls, whose values are needed but are not already in the return value of  $f$  on the un-incremented input. These are the additional values that should be computed together with  $f$ , if not already computed as intermediate results in  $f$ , and be stored together with the return value of  $f$ . This allows their values to be used in the computation on the incremented input, including maintenance of the corresponding values. For example, the computation of  $knap$  on the incremented

- 
- Step 1. Determine an input increment to  $f$ , yield operation  $next$  if this step succeeds
- 1.1 Identify all recursive calls to  $f$
  - 1.2 Represent their arguments in terms of the parameter  $x$  of  $f$
  - 1.3 Let  $prev(x)$  be the argument that changes minimally from  $x$
  - 1.4 Take the inverse  $next$  of  $prev$
- Step 2. Determine additional values to maintain, yield extended program  $fExt$
- 2.1 Identify function calls in  $f(next(x))$  not computed or returned by  $f(x)$
  - 2.2 Solve constraints for possible values of arguments to these function calls
  - 2.3 Construct program  $fExt$  that computes and returns the values of these function calls
- Step 3. Use and maintain cached values, yield incremental extended program  $fExt'$
- 3.1 Introduce  $fExt'$ , with value  $rExt$  of  $fExt(x)$  as a new parameter, to compute  $fExt(next(x))$
  - 3.2 Expand and simplify  $fExt(next(x))$
  - 3.3 Replace function calls in it with retrievals of their values from result  $rExt$  of  $fExt(x)$
- Step 4. Form optimized program, yield optimized  $fExt$  and  $f$
- 4.1 Construct base cases of the extended program  $fExt$
  - 4.2 Define  $fExtOpt$  to use  $fExt'$  repeatedly starting at the base cases constructed
  - 4.3 Define  $fOpt$  to retrieve the original return value from the value of  $fExtOpt$
- 

**Figure 1: An algorithm for optimizing recursive function  $f$ .**

---

input  $\langle i + 1, u \rangle$  is, by definition,

```

knap(i + 1, u)
= if i + 1 = 0 ∨ u ≤ 0 then 0
  else if w[i + 1] > u then knap(i, u)
  else max(v[i + 1] + knap(i, u - w[i + 1]), knap(i, u))

```

This needs  $knap(i, u)$  in the second branch and  $knap(i, u - w[i + 1])$  and  $knap(i, u)$  in the third branch. The value of  $knap(i, u)$  is just the return value of  $knap$  on the un-incremented input. The value of  $knap(i, u - w[i + 1])$ , however, is not computed by  $knap$  on the un-incremented input, and thus should be to be computed and cached together with  $knap(i, u)$ .

Among the values identified, those that are not computed at all as intermediate results in  $f$  are called auxiliary information. Computing them together with  $f$  means that there might potentially be extra computation, and thus in general explicit time and space analysis should be applied to determine whether this is worthwhile. However, since they are needed in computation on the incremented input, the rationale is that the cost can be amortized, and saving them to avoid potentially repeated computation of them can yield overall large speedup.

Next, determine appropriate data structures to use for caching the values identified above. In particular, for values of function calls whose arguments have a range of possible values, introduce arrays that map each possible argument value to the corresponding return value, as described below; values of other function calls are cached in components of tuples, as discussed in Section 4.

An argument of a function call may have a range of possible values if it depends on global variables whose values are independent of input sizes. For example,  $w[i]$ 's in the 0-1 knapsack problem are such global variables; their values are independent of the number of items or the total weight considered. The global variables may be constrained from

the problem specification. For example, the values of  $w[i]$ 's are positive integers. The definition of the function may also constrain the range of the argument for which the return value needs to be computed. For example,  $knap(i, u) = 0$  for  $u \leq 0$ . To obtain the range of the argument, simplify the argument expression together with constraints on the global variables. Once the range of the argument is determined, we extend  $f$  to  $fExt$  that computes and caches all possibly needed values in an array.

For example, the value of  $w[i + 1]$  identified above is a positive integer, and if  $w[i + 1] \geq u$ , then  $knap(i, u - w[i + 1]) = 0$  by definition. So we may only need the value of  $knap(i, u - w[i + 1])$  for any integer value of  $w[i + 1]$  from 1 to  $u - 1$ , i.e., any integer value of  $u - w[i + 1]$  from 1 to  $u - 1$ . That is, we may need  $knap(i, k)$  for  $k = 1..u - 1$ . Since the original return value is  $knap(i, u)$ , we cache  $knap(i, k)$  for  $k = 1..u$  in an auxiliary array.

$$knapExt(i, u) = \text{for } k := 1 \text{ to } u \text{ do} \\ rExt[k] := knap(i, k)$$

The original return value of  $knap(i, u)$  is then **if**  $u \leq 0$  **then** 0 **else**  $knapExt(i, u)[u]$ .

### 3.3 Step 3. Use and maintain cached values

This step transforms the extended program  $fExt$  on an incremented input  $next(x)$  to use the cached result  $rExt$  of  $fExt$  on un-incremented input  $x$ , and yields an incremental extended program  $fExt'(x, rExt)$ . First, introduce function  $fExt'(x, rExt)$  to compute  $fExt(next(x))$ , where  $rExt = fExt(x)$ . Then, expand  $fExt(next(x))$  by definition, and simplify operations in the resulting expression. Afterwards, replace function calls by retrievals of their values from the result  $rExt$  of  $fExt(x)$ . Details of these steps were studied earlier [24]. What's new here is the use and maintenance of values in indexed data structures. For the 0-1 knapsack

example, the transformations are as follows:

```

knapExt'(i, u, rExt)
= knapExt(i + 1, u)
    -- introduce knapExt'
= for k := 1 to u do
    rExt'[k] := knap(i + 1, k)
    -- expand knapExt by definition
= for k := 1 to u do
    rExt'[k] := if i + 1 = 0 ∨ u < 0 then 0
                else if w[i + 1] > u then knap(i, u)
                else max(v[i + 1] + knap(i, u - w[i + 1]),
                        knap(i, u))
    -- expand knap by definition and
    simplify i + 1 - 1 to 1

```

Afterwards, replace each *knap*(*i*, *k*) with retrieval *rExt*[*k*] from the result *rExt* of *knapExt*, yielding

```

knapExt'(i, u, rExt)
= for k := 1 to u do
    rExt'[k] := if i + 1 = 0 ∨ u < 0 then 0
                else if w[i + 1] > u then rExt[u]
                else max(v[i + 1] + rExt[u - w[i + 1]],
                        rExt[u])

```

### 3.4 Step 4. Form optimized program

This step has three substeps. First, construct the base cases of the extended function *fExt*. This constructs an array the same as the extended function, and then simplifies each element to be the corresponding base case of the original function under the base case condition. For the 0-1 knapsack problem, this first constructs for *k* := 1 to *u* do *rExt*[*k*] := *knap*(*i*, *k*), and then simplifies each *knap*(*i*, *k*) to 0 by definition of *knap* under the base condition *i* = 0.

Next, define the optimized extended function *fExtOpt*, which uses the base cases constructed above and calls itself repeatedly, each time computing incrementally by calling the incremental version *fExt'*. For the 0-1 knapsack problem, this yields

```

knapExtOpt(i, u)
= if i = 0 then
    -- base case
    for k := 1 to u do rExt[k] := 0
  else let rExt = knapExtOpt(i - 1, u) in
    knapExt'(i - 1, u, rExt)
    -- call incremental version

```

Finally, define the optimized function *fOpt* to retrieve the original return value from *fExtOpt*. For the 0-1 knapsack problem, this yields

```

knapOpt(i, u) = if u ≤ 0 then 0
                  else knapExtOpt(i, u)[u]

```

The time and space complexities of an optimized program that uses an incremental program is usually easy to analyze. For time complexity, computations that proceed in incremental fashions usually contribute linear factors independently. For the 0-1 knapsack problem, *knapOpt*(*n*, *W*) calls *knapExtOpt* once; *knapExtOpt* calls itself recursively and *knapExt'* repeatedly *n* times; *knapExt'* has a loop that iterates *O*(*W*) times, each time performing a constant number of primitive operations. Thus, *knapOpt*(*n*, *W*) takes *O*(*nW*) time. For space complexity, the size of the cached values contributes directly to the space consumption. For the 0-1 knapsack problem, this is the auxiliary array of size *O*(*W*). The *O*(*n*) stack space can be removed as discussed in Section 4.3. Textbook algorithms typically use a 2-dimensional array of size *O*(*nW*).

For the 0-1 knapsack problem, additional invariants may be employed so that linked lists may be used instead of arrays, by keeping items in order of increasing value and decreasing weight [2]. This achieves the same worst-case time and space complexities as ours but may take less actual time and space, depending on the particular input, due to sparsity of elements in the linked lists. Our method exploits only control and data structures in the given program; we do not yet know how to systematically discover stronger invariants such as orderings found in algorithms that use greedy or thinning strategies [6].

## 4. INDEXED VS. RECURSIVE DATA STRUCTURES

An alternative to indexed data structures for maintaining computed values is recursive data structures. Our same method of incrementalization for optimization is sufficiently general for exploiting also recursive data structures. This section describes a simple, new method for this based on selective caching. We use the same four steps described in Section 3, except that Step 2 is extended. Consider the binomial coefficient function *bin*(*n*, *k*), where 0 ≤ *k* ≤ *n*, defined by

```

bin(n, k) = if k = 0 ∨ k = n then 1
              else bin(n - 1, k - 1) + bin(n - 1, k)

```

Step 1 determines the input increment of *bin* to be *next*(*n*, *k*) = ⟨*n* + 1, *k*⟩.

Step 2 determines additional values, if any, that need to be computed and returned together with *f* so that they can be used to compute *f*(*next*(*x*)). For the binomial coefficient function, we have

```

bin(n + 1, k) = if k = 0 ∨ k = n + 1 then 1
                  else bin(n, k - 1) + bin(n, k)

```

There are two function calls. The value of *bin*(*n*, *k*) is just the original return value. The value of *bin*(*n*, *k* - 1) is not computed in *bin*(*n*, *k*), so it incurs the need to extend *bin* to return additional values. What additional values?

The general algorithm for Step 2 determines, for each function call whose value is not in the return value of *f*(*x*), the set of all possible values the argument of the call can take.

When this set is a singleton, then simply extend the original function to return this single value as a component of a tuple that contains also the original value. When this set has more than one element, there are two possible cases.

In one case, the argument depends on global variables whose values are independent of input sizes, as in Section 3. In this case, the range of the argument is determined by solving constraints on the variables involved, and an indexed data structure is introduced for all possible results, as discussed in Section 3.

In the other case, the argument is uniquely determined by input sizes, however, computing the function on this argument recursively incurs the need to cache the function value on another argument. In this case, one may still determine all possible arguments by solving constraints and use an array, as described below in Section 4.2. However, the use of array is not necessary, and we may make use of recursion to reduce the need of caching an array of values, by caching only individual values in components of tuples, and make use of recursion to form recursive data structures, as described below in Section 4.1.

## 4.1 Using recursive data structures

The goal is to compute  $bin(n+1, k)$  using  $bin(n, k)$ . By definition of  $bin$ , computing  $bin(n+1, k)$  needs  $bin(n, k-1)$ , which is not computed in  $bin(n, k)$ . However, computing  $bin(n+1, k)$  using  $bin(n, k)$  means computing  $bin(n, k-1)$  using  $bin(n-1, k-1)$ , which is computed in  $bin(n, k)$ . This means that we should extend  $bin(n, k)$  to return  $bin(n-1, k-1)$  recursively, and use it in recursively computing  $bin(n+1, k)$ . This extension yields:

$$binExt(n, k) = \text{if } k = 0 \vee k = n \text{ then } \langle 1 \rangle \\ \text{else let } v = binExt(n-1, k-1) \text{ in} \\ \langle 1st(v) + bin(n-1, k), v \rangle$$

where  $binExt(n, k)$  recursively computes  $binExt(n-1, k-1)$  and puts this value as the second component of a tuple. To ensure that the original return value is always in the first component of a tuple, the base case is extended as well to return a singleton tuple.

Step 3 transforms  $binExt(n+1, k)$  to use the value  $rExt$  of  $binExt(n, k)$ , as below. Introduction of the condition  $k = n$  is determined by the condition  $k = n$  in the original function  $bin(n, k)$  and the fact that  $2nd(rExt)$  is returned only in the false branch of  $k = n$ . A detailed method for the

transformations was studied earlier [24]. We obtain

$$binExt'(n, k, rExt) \\ = binExt(n+1, k) \\ \quad \quad \quad \text{-- introduce } binExt' \\ = \text{if } k = 0 \vee k = n+1 \text{ then } \langle 1 \rangle \\ \quad \text{else if } k = n \text{ then} \\ \quad \quad \text{let } v = binExt(n, k-1) \text{ in} \\ \quad \quad \langle 1st(v) + 1st(binExt(n, k)), v \rangle \\ \quad \quad \text{else let } v = binExt(n, k-1) \text{ in} \\ \quad \quad \langle 1st(v) + 1st(binExt(n, k)), v \rangle \\ \quad \quad \quad \text{-- expand } binExt \text{ by definition and} \\ \quad \quad \quad \text{introduce condition } k = n \text{ from } bin(n, k) \\ = \text{if } k = 0 \vee k = n+1 \text{ then } \langle 1 \rangle \\ \quad \text{else if } k = n \text{ then} \\ \quad \quad \text{let } v = binExt'(n-1, k-1, \langle \rangle) \text{ in} \\ \quad \quad \langle 1st(v) + 1, v \rangle \\ \quad \quad \text{else let } v = binExt'(n-1, k-1, 2nd(rExt)) \text{ in} \\ \quad \quad \langle 1st(v) + 1st(rExt), v \rangle \\ \quad \quad \quad \text{-- replace calls to } binExt \text{ by retrievals or} \\ \quad \quad \quad \text{calls to the incremental version } binExt'$$

Step 4 forms the optimized program

$$binOpt(n, k) = 1st(binExtOpt(n, k)) \\ binExtOpt(n, k) = \text{if } k = 0 \vee k = n \text{ then } \langle 1 \rangle \\ \quad \text{else let } rExt = binExtOpt(n-1, k) \text{ in} \\ \quad \quad binExt'(n-1, k, rExt)$$

The time complexity of  $binOpt(n, k)$  is  $O((n-k)*k)$ , since it calls  $binExtOpt(n, k)$  once;  $binExtOpt(n, k)$  recursively calls itself and  $binExt'$  a total of  $O(n-k)$  times;  $binExt'$  recursively calls itself  $O(k)$  times, each computing a few primitive operations. The space complexity is the size of the cached list of intermediate results, which is  $O(k)$ . The  $O(n-k)$  stack space can be eliminated as discussed in Section 4.3.

## 4.2 Using indexed data structures

Now we optimize the function  $bin$  using indexed data structures and compare with the result from Section 4.1. Computing  $bin(next(n, k))$ , i.e.,  $bin(n+1, k)$ , needs the value of  $bin(n, k-1)$ , so we need to extend  $bin(n, k)$  to return this value; but then computing  $bin(next(n, k-1))$ , i.e.,  $bin(n+1, k-1)$ , will need the value of  $bin(n, k-2)$ ; this may repeat until the base case of  $k = 0$  is reached. To determine the set  $S$  of possible arguments, collect constraints as follows:  $\langle n, k \rangle \in S$ , and if  $\langle n, j \rangle \in S$  and  $j \geq 0$  then  $\langle n, j-1 \rangle \in S$ . Solving the constraints, one obtains  $S = \{\langle n, j \rangle : j = 0, \dots, k\}$ . So Step 2 yields the following extended program:

$$binExt(n, k) = \text{for } j := 0 \text{ to } k \text{ do} \\ \quad rExt[j] := bin(n, j)$$

Step 3 transforms  $binExt(n+1)$  to use the result  $rExt$  of

$binExt(n)$ .

```

binExt'(n, k, rExt)
= binExt(n + 1)
= for j := 0 to k do
  rExt'[j] := bin(n + 1, j)
= for j := 0 to k do
  rExt'[j] := if k = 0 ∨ k = n then 1
              else bin(n, k - 1) + bin(n, k)
= for j := 0 to k do
  rExt'[j] := if k = 0 ∨ k = n then 1
              else rExt[k - 1] + rExt[k]

```

Step 4 forms the optimized program

```

binOpt(n, k) = binExtOpt(n, k)[k]
binExtOpt(n, k) = if k = 0 ∨ k = n then
                  for j := 0 to k do rExt[j] := 1
                  else let rExt = binExtOpt(n - 1, k) in
                       binExt'(n - 1, k, rExt)

```

The time and space complexities of  $binOpt(n, k)$  are exactly the same as those in Section 4.1, except that  $binExt'$  has a loop that iterates, rather than recursively calls itself,  $O(k)$  times, each iteration computing a few primitive operations, and that  $binExt'$  uses an array, rather than a list, of size  $O(k)$ .

### 4.3 Summary and discussion

Our method for optimizing recursive functions can use both indexed and recursive data structures. It can use arrays when needed, can derive versions that use arrays and linked lists when both are possible, and can use the theoretically less powerful linked lists to achieve the same asymptotic running time in many cases. This makes it easier to compare the uses of different data structures.

The essence of our method is that what and how to cache are driven by the input increment operation. For example, the 0-1 knapsack problem computes incrementally in considering each next item as the input increment, and to do so needs an 1-dimension array for the weight dimension.

The incremental fashion of computation actually yields an iterative computation for a recursive function. For example, for the 0-1 knapsack problem, if we extend our language, we could use a **while**-loop and directly form optimized  $knapExt$  as

```

knapExt(i, u)
= j := 0; for k := 1 to u do rExt[k] := 0;
  while j < i do
    j := j + 1; rExt := knapExt'(j - 1, u, rExt);
  return rExt;

```

This removes the time and space overhead of the call stack, reducing the space complexity of  $knapOpt$  from  $O(n + W)$  to  $O(W)$ . Furthermore, this optimized version enables one

to easily see that an additional array could be used for copying  $rExt$ , removing the need to allocate and garbage-collect the array for each incremental step. We have preliminary results on transforming recursion into iteration using incrementalization [20]. Detailed study is a future work.

When there are multiple minimal increment operations, for example, the longest common subsequence and matrix chain multiplication problems [12, 19] each has two, we found that any one of them may be used, and they may lead to optimized programs with the same time and space complexity.

## 5. ADDITIONAL EXAMPLES AND EXPERIMENTS

This section discusses shortest path problems and summarizes other examples and experiments.

### 5.1 Single-source shortest paths

Given a directed weighted graph and a vertex  $s$  in the graph, the single-source shortest path problem [12, Chapter 25] is to compute, for each vertex  $t$  in the graph, the minimum weight from  $s$  to  $t$  following a path in the graph. There is a well-known simple recursion that computes the shortest distance from vertex  $u$  to vertex  $v$  as  $d(u, v, n - 1)$ , where  $n$  is the number of vertices in the graph, and  $d(i, j, m)$ , the minimum weight of any path from vertex  $i$  to vertex  $j$  that contains at most  $m$  edges, is defined as, for  $m \geq 0$ ,

$$d(i, j, 0) = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{otherwise} \end{cases}$$

$$d(i, j, m) = \min_{1 \leq k \leq n} \{d(i, k, m - 1) + w_{kj}\} \quad \text{for } m \geq 1$$

Absence of an edge  $\langle j, k \rangle$  is modeled by taking  $w_{jk} = \infty$ . The single-source shortest path problem is to compute  $d(s, t, n - 1)$  for all  $t$  from 1 to  $n$ . A straightforward solution is

```

sssp(s) = for t := 1 to n do
          a[t] := d(s, t, n - 1)
d(i, j, m) = if m = 0 then
              if i = j then 0 else ∞
              else dsub(i, j, 1, m)
dsub(i, j, k, m) = let s = d(i, k, m - 1) + w[k][j] in
                  if k = n then s
                  else let min = dsub(i, j, k + 1, m) in
                       if s < min then s else min

```

where  $n$  and  $w$  are implicit parameters. Mutual recursion is present, but the same transformations in Section 3 can be used. The function  $d$  is called recursively through  $dsub$  and takes exponential time. It can be optimized using our method as follows.

Step 1 identifies appropriate input increment for the expensive recursive function  $d$ . Arguments of  $d$  are  $\langle i, j, m \rangle$ , and arguments of recursive calls to  $d$  are  $\langle i, k, m - 1 \rangle$  for  $k = 1..n$ . Clearly, the recursive call to  $d$  with arguments  $\langle i, j, m - 1 \rangle$  differs least from  $\langle i, j, m \rangle$ . Thus the minimum increment operation is  $next(i, j, m) = \langle i, j, m + 1 \rangle$ .

Step 2 determines appropriate values to cache and data

structures to use for incremental computation. By definition, in the computation of  $d(i, j, m + 1)$ , values  $d(i, k, m)$  for  $k = 1..n$  are needed. This is still so considering that the values of  $d(s, t, n - 1)$  on all  $t$  from 1 to  $n$  are needed in  $sssp(s)$ . Caching these values in an array  $rExt$  where  $rExt[j] = d(i, j, m)$  for  $j = 1..n$ , we obtain

$$dExt(i, m) = \text{for } j := 1 \text{ to } n \text{ do } rExt[j] := d(i, j, m)$$

The original return value of  $d(i, j, m)$  is then  $dExt(i, m)[j]$ .

Step 3 incrementalizes  $dExt(i, m)$  under the increment operation from Step 1, i.e., transforms  $dExt(i, m + 1)$  to use  $rExt = dExt(i, m)$ . This step yields

$$\begin{aligned} dExt'(i, m, rExt) &= \text{for } j := 1 \text{ to } n \text{ do } rExt'[j] := d'(i, j, m, rExt) \\ d'(i, j, m, rExt) &= dsub'(i, j, 1, m, rExt) \\ dsub'(i, j, k, m, rExt) &= \text{let } s = rExt[k] + w_{kj} \text{ in} \\ &\quad \text{if } k = n \text{ then } s \\ &\quad \text{else let } min = dsub'(i, j, k + 1, m, rExt) \text{ in} \\ &\quad \text{if } s < min \text{ then } s \text{ else } min \end{aligned}$$

Step 4 forms the optimized program.  $dExtOpt$  uses  $dExt'$  to compute incrementally.  $dOpt$  retrieves its value from that of  $dExtOpt$ .  $ssspOpt$  calls  $dOpt$  instead of  $d$ .

$$\begin{aligned} dOpt(i, j, m) &= dExtOpt(i, m)[j] \\ dExtOpt(i, m) &= \text{if } m = 0 \text{ then} \\ &\quad \text{for } j := 1 \text{ to } n \text{ do} \\ &\quad \quad rExt[j] := \text{if } i = j \text{ then } 0 \text{ else } \infty \\ &\quad \text{else let } rExt = dExtOpt(i, m - 1) \text{ in} \\ &\quad \quad dExt'(i, m - 1, rExt) \end{aligned}$$

A final cleanup, not discussed in Section 3 and currently done manually, can remove the unnecessary copy of the return value of  $dExtOpt$  into the return value of  $ssspOpt$ , and the unnecessary intermediate functions  $dOpt$  and  $d'$ , yielding

$$\begin{aligned} ssspOpt(s) &= dExtOpt(s, n - 1) \\ dExt'(i, m, rExt) &= \text{for } j := 1 \text{ to } n \text{ do} \\ &\quad rExt'[j] := dsub'(i, j, 1, m, rExt) \end{aligned}$$

The function  $ssspOpt$  takes  $O(n^3)$  time. This is because  $dExtOpt(s, n - 1)$  calls itself recursively and calls  $dExt'$   $O(n)$  times; each  $dExt'$  calls  $dsub'$  iteratively  $O(n)$  times; each  $dsub'$  calls itself  $O(n)$  times, each time performing a constant number primitive operations.

If one strengthens the original recursive definitions by considering only  $k$ 's such that  $w_{kj} \neq \infty$ , then in the resulting optimized program, for each  $j$  from 1 to  $n$ , only its predecessor nodes, not all  $n$  nodes in the graph, are considered. Suppose the total number of edges in the given graph is  $e$ , the optimized program takes  $O(ne)$  time. This is exactly the Bellman-Ford algorithm [12] without the triangle inequality

test for each edge at the end. It is the best known algorithm that works on general graphs with arbitrary weights.

## 5.2 All-pairs shortest paths

Given a directed weighted graph, the all-pairs shortest path problem [12, Chapter 26] is to compute, for each pair of nodes  $i$  and  $j$  in the graph, the shortest distance from  $i$  to  $j$ . This can be defined as

$$\begin{aligned} apsp() &= \text{for } s := 1 \text{ to } n \text{ to} \\ &\quad a[s] := \text{for } t := 1 \text{ to } n \text{ do} \\ &\quad \quad b[t] := d(s, t, n - 1) \end{aligned}$$

where  $d$  is as in Section 5.1. The derivation of  $dExtOpt$  proceeds exactly as in Section 5.1. The final optimized program is

$$\begin{aligned} apspOpt() &= \text{for } s := 1 \text{ to } n \text{ to} \\ &\quad a[s] := dExtOpt(s, n - 1) \end{aligned}$$

$apsp$  takes exponential time.  $apspOpt$  takes a factor of  $O(n)$  more time than the algorithms in Section 5.1, so it is  $O(n^4)$ , or  $O(n^2e)$  if one considers only predecessor nodes. A faster algorithm can be derived as follows.

There is another recursion that computes the shortest distance from vertex  $u$  to vertex  $v$  as  $d(u, v, n)$ , where  $d(i, j, m)$  is the minimum weight of any path from  $i$  to  $j$  with intermediate nodes in  $\{1, 2, \dots, m\}$  and is defined as, for  $m \geq 0$ ,

$$d(i, j, m) = \begin{cases} w_{ij} & \text{if } m = 0 \\ \min(d(i, j, m - 1), & \text{otherwise} \\ \quad d(i, m, m - 1) + d(m, j, m - 1)) & \end{cases}$$

The all-pairs shortest path problem is to compute  $d(s, t, n)$  for all  $s$  and  $t$  from 1 to  $n$ . A straightforward solution is

$$\begin{aligned} apsp() &= \text{for } s := 1 \text{ to } n \text{ do} \\ &\quad a[s] := \text{for } t := 1 \text{ to } n \text{ do } b[t] := d(s, t, n) \\ d(i, j, m) &= \text{if } m = 0 \text{ then } w[i][j] \\ &\quad \text{else let } min = d(i, j, m - 1) \text{ in} \\ &\quad \quad \text{let } viam = d(i, m, m - 1) + d(m, j, m - 1) \text{ in} \\ &\quad \quad \text{if } viam < min \text{ then } viam \text{ else } min \end{aligned}$$

where  $n$  and  $w$  are implicit parameters. Again, the function  $d$  takes exponential time and can be optimized using our method as follows.

Step 1 identifies the minimum increment operation for  $d$  as  $next(i, j, m) = \langle i, j, m + 1 \rangle$ , since there is a recursive call with argument  $\langle i, j, m - 1 \rangle$  that differs least from  $\langle i, j, m \rangle$ .

Step 2 first determines that  $d(i, j, m + 1)$  needs the values of  $d(i, m, m)$  and  $d(m, j, m)$  in addition to  $d(i, j, m)$ . Furthermore, given that the values  $d(s, t, n)$  on all  $s$  and  $t$  from 1 to  $n$  are needed in  $apsp()$ , the values of  $d(i, j, m + 1)$  for all  $i$  and  $j$  from 1 to  $n$  need the values of  $d(i, j, m)$  for all  $i$  and  $j$  from 1 to  $n$  as well. Caching all needed values in an array

$rExt$  of arrays  $r1Ext$  yields

```

dExt(m) = for i := 1 to n do
           rExt[i] := for j := 1 to n do
                       r1Ext[j] := d(i, j, m)

```

The original return value of  $d(i, j, m)$  is  $dExt(m)[i][j]$ .

Step 3 transforms  $dExt(m + 1)$  to use  $rExt = dExt(m)$ . This yields

```

dExt'(m, rExt)
= for i := 1 to n do
  rExt'[i] := for j := 1 to n do
              r1Ext'[j] := d'(i, j, m, rExt)

d'(i, j, m, rExt)
= if m = 0 then w[i][j]
  else let min = rExt[i][j] in
        let viam = rExt[i][m] + rExt[m][j] in
        if viam < min then viam else min

```

Step 4 forms the optimized program.  $dExtOpt$  uses  $dExt'$  to compute incrementally.  $dOpt$  retrieves its value from that of  $dExtOpt$ .  $apspOpt$  calls  $dOpt$  instead of  $d$ .

```

dOpt(i, j, m) = dExtOpt(m)[i][j]
dExtOpt(m) = if m = 0 then
              for i := 1 to n do
                rExt'[i] := for j := 1 to n do
                            r1Ext[j] := w[i][j]
              else let rExt = dExtOpt(m - 1) in
                    dExt'(m - 1, rExt)

```

Removing the unnecessary copy of the return value of  $dExtOpt$  into the return value of  $apspOpt$  yields

$$apspOpt() = dExtOpt(n)$$

The function  $apspOpt$  takes  $O(n^3)$  time. This is because  $dExtOpt(n)$  calls itself recursively and calls  $dExt' O(n)$  times; each  $dExt'$  calls  $d'$  iteratively  $O(n^2)$  times, each time performing a constant number of primitive operations. This is the Floyd-Warshall algorithm [12].

### 5.3 Other examples and experiments

Figure 2 summarizes some other examples that our method can optimize. All of them can be optimized with our method for using indexed data structures. The first ten can also be optimized with our method for using recursive data structures, either by caching selectively as in this paper, or by caching all and pruning as done previously [19], but the former is simpler. Some of these examples can be further improved by employing stronger invariants such as those used in greedy algorithms. We do not yet know how to discover stronger invariants systematically.

We have a semi-automatic system, CACHET, that helps us apply most tedious transformations automatically. The

parts that are left manual are either just because of incomplete implementation (mainly for parts that are easy to do by hand, e.g., taking minimals and inverses) or left so on purpose (for easy experimentation and demonstration, e.g., a unique enabled button needs to be pressed to continue after each major transformation). We believe that a fully automated system, with simple heuristics for taking minimals and inverses and with automated simplifiers for equality analysis, can derive all these examples and many more.

## 6. RELATED WORK AND CONCLUSION

Comparison with previous work on deriving efficient algorithms by transforming recursive functions or set-based languages appears in the introduction. There has also been work on deriving recursive equations from input and output relations, e.g., [35].

Techniques such as magic set and tabled Prolog, for bottom up evaluation of logic programs [3, 15, 25, 26, 36], are also closely related. Our method based on incrementalization essentially achieves bottom up evaluation by static transformation of the original program. It improves running time and at the same time minimizes space consumption. For example, for the simple Fibonacci function, our transformed program uses constant space, while a program using magic sets uses linear space.

Compared with our previous work on incrementalization and program optimization [18, 19], this work contains three substantial improvements. First, determining increment operations is made the single most important basis of the method, which drives what and how to cache, how to maintain them, and how to form an optimized program. Also, it is made as simple as determining a minimal change, rather than having to cover all recursive calls [19]. It is equally powerful for all real problems we've encountered; our previous algorithm was designed to handle more general cases, but so far they arise only in contrived examples.

Second, the use of indexed data structures was not studied in our previous methods [19]. This work generalizes them to use both indexed and recursive data structures in one unified method. Use of indexed data structures is essential; recursive data structures could lead to linear-time random access, and could use twice as much space and be several times slower to process, possibly asymptotically (even exponentially [5]) when interacting badly with garbage collection. Also, use of arrays often allows much simpler derivation.

Third, our previous methods use cache-and-prune [19, 23]: cache all possibly needed values, incrementalize to use these values, and prune unused values. The method in this paper selectively caches values by analyzing what's needed in the incremental computation. This analysis enables us to determine the use of arrays or linked lists that was impossible before. It also makes the incrementalization step simpler and removes the need to prune.

The main contribution of this work is the systematic method for optimizing recursive functions that unifies and generalizes previous methods. We have omitted details of some analyses and transformations, because they are mostly straight-

Examples	original program's running time	optimized prog's running time
Fibonacci function [29]	$O(2^n)$	$O(n)$
binomial coefficients [29]	$O(2^n)$	$O(n * k)$
longest common subsequence [12]	$O(2^{n+m})$	$O(n * m)$
matrix-chain multiplication [12]	$O(n * 3^n)$	$O(n^3)$
string editing distance [33]	$O(3^{n+m})$	$O(n * m)$
dag path sequence [8]	$O(2^n)$	$O(n^2)$
optimal polygon triangulation [12]	$O(n * 3^n)$	$O(n^3)$
optimal binary search tree [1]	$O(n * 3^n)$	$O(n^3)$
paragraph formatting [12]	$O(n * 2^n)$	$O(n^2)$
paragraph formatting 2 [19]	$O(n * 2^n)$	$O(n * width)$
0-1 knapsack [12]	$O(2^n)$	$O(n * weight)$
context-free-grammar parsing [1]	$O(n * (2 * size + 1)^n)$	$O(n^3 * size)$
single-source shortest paths [12]	$O(n^{n+1})$	$O(n * e)$
single-pair shortest paths [12]	$O(n^n)$	$O(n * e)$
all-pairs shortest paths [12]	$O(n^2 * 3^n)$	$O(n^3)$

Figure 2: Summary of Examples.

forward and can use existing methods, e.g., [9, 22, 23, 24].

An important open problem is a precise characterization and analysis of the effect of our optimization method, for predicting the complexity of the optimized program. A possible approach may be similar to recent work by McAllester [25] and Ganzinger [15] for logic programs.

## Acknowledgments

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

## 7. REFERENCES

- [1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1974.
- [2] J. H. Ahrens and G. Finke. Merging and sorting applied to the 0-1 knapsack problem. *Operations Research*, 23(6):1099–1109, 1975.
- [3] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 1–16. ACM, New York, 1986.
- [4] F. L. Bauer and H. Wössner. *Algorithmic Language and Program Development*. Springer-Verlag, Berlin, 1982.
- [5] H. Bernstein. To transpose or not to transpose—performance issues in the use of Java-based XML software for real-world problems. <http://www.cs.sunysb.edu/~liu/darseminar/msg00005.html>, Sept. 2001.
- [6] R. Bird and O. de Moor. *Algebra of Programming*. Prentice-Hall, 1996.
- [7] R. S. Bird. Tabulation techniques for recursive programs. *ACM Comput. Surv.*, 12(4):403–417, Dec. 1980.
- [8] R. S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Trans. Program. Lang. Syst.*, 6(4):487–504, Oct. 1984.
- [9] 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, June 1997.
- [10] W.-N. Chin and S.-C. Khoo. Tupling functions with multiple recursion parameters. In P. Cousot, M. Falaschi, G. Filè, and A. Rauzy, editors, *Proceedings of the 3rd International Workshop on Static Analysis*, volume 724 of *Lecture Notes in Computer Science*, pages 124–140. Springer-Verlag, Berlin, Sept. 1993.
- [11] N. H. Cohen. Eliminating redundant recursive calls. *ACM Trans. Program. Lang. Syst.*, 5(3):265–299, July 1983.
- [12] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press/McGraw-Hill, 1990.
- [13] O. de Moor. A generic program for sequential decision processes. In M. Hermenegildo and D. S. Swierstra, editors, *Programming Languages: Implementations, Logics, and Programs*, volume 982 of *Lecture Notes in Computer Science*, pages 1–23. Springer-Verlag, Berlin, 1995.
- [14] D. P. Friedman, D. S. Wise, and M. Wand. Recursive programming through table look-up. In *Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation*, pages 85–89. ACM, New York, 1976.

- [15] H. Ganzinger and D. McAllester. A new meta-complexity theorem for bottom-up logic programs. In *Proceedings of the International Joint Conference on Automated Reasoning*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2001.
- [16] H. Khoshnevisan. Efficient memo-table management strategies. *Acta Informatica*, 28(1):43–81, 1990.
- [17] N. Klarlund. *MONA Version 1.3 User Manual*, Oct. 1998.
- [18] Y. A. Liu. Efficiency by incrementalization: An introduction. *Higher-Order and Symbolic Computation*, 13(4):289–313, Dec. 2000.
- [19] 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.
- [20] 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, Jan. 2000.
- [21] Y. A. Liu and S. D. Stoller. Optimizing Ackermann's function by incrementalization. Technical Report DAR 01-1, Computer Science Department, SUNY Stony Brook, Jan. 2001.
- [22] 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.
- [23] 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.
- [24] Y. A. Liu and T. Teitelbaum. Systematic derivation of incremental programs. *Sci. Comput. Program.*, 24(1):1–39, Feb. 1995.
- [25] D. McAllester. On the complexity analysis of static analyses. In *Proceedings of the 6th International Static Analysis Symposium*, volume 1694 of *Lecture Notes in Computer Science*, pages 312–329. Springer-Verlag, Berlin, Sept. 1999.
- [26] J. F. Naughton and R. Ramakrishnan. Bottom-up evaluation of logic programs. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 640–700. The MIT Press, Cambridge, Mass., 1991.
- [27] 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.
- [28] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst.*, 4(3):402–454, July 1982.
- [29] H. A. Partsch. *Specification and Transformation of Programs—A Formal Approach to Software Development*. Springer-Verlag, Berlin, 1990.
- [30] 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.
- [31] A. Pettorossi and M. Proietti. Program derivation via list introduction. In R. Bird and L. Meertens, editors, *Algorithmic Languages and Calculi*. Chapman & Hall, London, U.K., 1997.
- [32] W. Pugh. The Omega Test: A fast and practical integer programming algorithm for dependence analysis. *Commun. ACM*, 31(8):102–114, Aug. 1992.
- [33] P. W. Purdom and C. A. Brown. *The Analysis of Algorithms*. Holt, Rinehart and Winston, 1985.
- [34] 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.
- [35] D. R. Smith. Structure and design of problem reduction generators. In B. Möller, editor, *Constructing Programs from Specifications*, pages 91–124. North-Holland, Amsterdam, 1991.
- [36] D. S. Warren. The XSB tabled logic programming system (abstract). In J. Minker, editor, *Workshop on Logic-Based Artificial Intelligence, Washington, DC, June 14–16, 1999*, College Park, Maryland, 1999. Computer Science Department, University of Maryland.
- [37] A. Webber. Optimization of functional programs by grammar thinning. *ACM Trans. Program. Lang. Syst.*, 17(2):293–330, Mar. 1995.