Dynamic Programming via Static Incrementalization

Yanhong A. Liu* and Scott D. Stoller*

November 18, 2002

Abstract

Dynamic programming is an important algorithm design technique. It is used for problems whose solutions involve recursively solving subproblems that share subsubproblems. While a straightforward recursive program solves common subsubproblems repeatedly, a dynamic programming algorithm solves every subsubproblem just once, saves the result, and reuses it when the subsubproblem is encountered again. This can reduce the time complexity from exponential to polynomial. This paper describes a systematic method for transforming programs written as straightforward recursions into programs that use dynamic programming. The method extends the original program to cache all possibly computed values, incrementalizes the extended program with respect to an input increment to use and maintain all cached results, prunes out cached results that are not used in the incremental computation, and uses the resulting incremental program to form an optimized new program. Incrementalization statically exploits semantics of both control structures and data structures and maintains as invariants equalities characterizing cached results. It provides the basis of a general method for achieving drastic program speedups. Compared with previous methods that perform memoization or tabulation, the method based on incrementalization is more powerful and systematic. It has been implemented in a prototype system CACHET and applied to numerous problems and succeeded on all of them.

1 Introduction

Dynamic programming is an important technique for designing efficient algorithms [2, 15, 52]. It is used for problems whose solutions involve recursively solving subproblems that share subsubproblems. While a straightforward recursive program solves common subsubproblems repeatedly, a dynamic programming algorithm solves every subsubproblem just once, saves the result in a table, and reuses the result when the subsubproblem is encountered again. This can reduce the time complexity from exponential to polynomial. The technique is generally applicable to all problems that can be solved efficiently by memoizing results of subproblems [4, 5].

Given a straightforward recursion, there are two traditional ways to achieve the effect of dynamic programming [15]: memoization [40] and tabulation [5].

*This work is supported in part by ONR under grants N00014-99-1-0132 and N00014-99-1-0358 and by NSF under grants CCR-9711253 and CCR-9876058. This article is a revised and extended version of a paper that appeared in *Proceedings of the 8th European Symposium on Programming*, Amsterdam, The Netherlands, March 1999. Authors' address: Computer Science Department, State University of New York at Stony Brook, Stony Brook, NY 11794. Phone: 631-632-{8463,1627}. Email: {liu,stoller}@cs.sunysb.edu.

Memoization uses a mechanism that is separate from the original program to save the result of each function call or reduction as the program executes [1, 19, 20, 25, 27, 28, 40, 41, 45, 49, 51]. The idea is to keep a separate table of solutions to subproblems, modify recursive calls to first look up in the table, and then, if the subproblem has been computed, use the saved result, otherwise, compute it and save the result in the table. This method has two advantages. First, the original recursive program needs virtually no change. The underlying interpretation mechanism takes care of table filling and lookup. Second, only values needed by the original program are actually computed, which is optimal in a sense. Memoization has two disadvantages. First, the mechanism for table filling and lookup has an interpretive overhead. Second, no general strategy for table management is efficient for all problems.

Tabulation statically determines what shape of table is needed to store the values of all possibly needed subcomputations, introduces appropriate data structures for the table, and computes the table entries in a bottom-up fashion so that the solution to a superproblem is computed using available solutions to subproblems [5, 11, 12, 13, 14, 24, 45, 46, 47, 48]. This overcomes both disadvantages of memoization. First, table filling and lookup are compiled into the resulting program, so no separate mechanism is needed for the execution. Second, strategies for table filling and lookup can be specialized to be efficient for particular problems. However, tabulation has two drawbacks. First, it usually requires a thorough understanding of the problem and a complete manual rewrite of the program [15]. Second, to statically ensure that all values possibly needed are computed and stored, a table that is larger than necessary is often used; it may also include solutions to subproblems not actually needed in the original computation.

This paper presents a powerful method that statically analyzes and transforms straightforward recursive programs to efficiently cache and use the results of needed subproblems at appropriate program points in appropriate data structures. The method has three steps: (1) extend the original program to cache all possibly computed values, (2) incrementalize the extended program, with respect to an input increment, to use and maintain all cached results, (3) prune out cached results that are not used in the incremental computation, and use the resulting incremental program to form an optimized program. The method overcomes both drawbacks of tabulation. First, it consists of static program analyses and transformations that are general and systematic. Second, it stores only values that are necessary for the optimization; it also shows exactly when and where subproblems not in the original computation have to be included.

Our method is based on a number of static analyses and transformations studied previously by others [6, 9, 21, 42, 47, 55, 56, 62] and ourselves [30, 37, 38, 39] and improves them. Each of the caching, incrementalization, and pruning steps is simple, automatable, and efficient and has been implemented in a prototype system, CACHET. The system has been used in optimizing many programs written as straightforward recursions, including all dynamic programming problems found in [2, 15, 52], most in semi-automatic mode and some in fully automatic mode. Performance measurements confirm drastic asymptotic speedups.

The rest of the paper is organized as follows. Section 2 formulates the problem. Sections 3, 4,

and 5 describe the three steps. Section 6 summarizes and discuses related issues. Section 7 presents the experimentation and performance measurements. Section 8 compares with related work and concludes.

2 Formulating the problem

Straightforward solutions to many combinatorics and optimization problems can be written as simple recursions [52, 15]. For example, the matrix-chain multiplication problem [15, pages 302-314] computes the minimum number of scalar multiplications needed by any parenthesization in multiplying a chain of n matrices, where matrix i has dimensions $p_{i-1} \times p_i$. This can be computed as $m(1, n)$, where $m(i, j)$ computes the minimum number of scalar multiplications for multiplying matrices i through j and can be defined as: for $i \leq j$,

$$m(i,j) = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k \leq j-1}\{m(i,k) + m(k+1,j) + p_{i-1} * p_k * p_j\} & \text{otherwise} \end{cases}$$

The longest-common-subsequence problem [15, pages 314–320] computes the length $c(n, m)$ of the longest common subsequence of two sequences $\langle x_1, x_2, ..., x_n\rangle$ and $\langle y_1, y_2, ..., y_m\rangle$, where $c(i, j)$ can be defined as: for $i, j \geq 0$,

$$c(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c(i-1,j-1)+1 & \text{if } i \neq 0 \text{ and } j \neq 0 \text{ and } x_i = y_j \\ \max(c(i,j-1), c(i-1,j)) & \text{otherwise} \end{cases}$$

Both of these examples are literally copied from the textbook by Cormen, Leiserson, and Rivest [15].

These 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

$$f(v_1, ..., v_n) \triangleq e$$

where an expression e is given by the grammar

$e ::= v$		variable
$\mid$	$c(e_1, ..., e_n)$	constructor application
$\mid$	$p(e_1, ..., e_n)$	primitive function application
$\mid$	$f(e_1, ..., e_n)$	function application
$\mid$	**if** e_1 **then** e_2 **else** e_3	conditional expression
$\mid$	**let** $v = e_1$ **in** e_2	binding expression

We include arrays as variables and use them for indexed access such as x_i and p_j above. For convenience, we allow global variables, i.e., variables that do not change across function calls, to be implicit parameters to functions; such variables can be identified easily for our language even if they are given as explicit parameters. For a conditional expression whose condition depends on a global variable, we assume that both branches may be executed without divergence or run-time errors regardless of the value of the condition, which holds for the large class of combinatorics and optimization problems we handle.

Figure 1 gives programs for the examples above. Invariants about an input are not part of a program but are written explicitly to be used by the transformations. Clearly, x and y are implicit parameters to c, and p is an implicit parameter to m and $msub$. Condition $x[i] = y[j]$ in c depends on global variables x and y. These examples do not use data constructors, but our previous papers contain a number of examples that use them [37, 38, 39] and our method handles them.

$$
\begin{array}{l}
c(i,j) \qquad \text{where } i,j \geq 0 \\
\triangleq \ \textbf{if } i = 0 \ \vee j = 0 \textbf{ then } 0 \\
\quad \textbf{else if } x[i] = y[j] \textbf{ then } c(i-1, j-1) + 1 \\
\quad \textbf{else } \max(c(i, j-1), c(i-1, j))
\end{array}
$$

$$
\begin{array}{ll}
m(i,j) \qquad \text{where } i \leq j & msub(i,j,k) \qquad \text{where } i \leq k \leq j-1 \\
\triangleq \ \textbf{if } i = j \textbf{ then } 0 & \triangleq \ \textbf{let } s = m(i,k) + m(k+1, j) + p[i-1] * p[k] * p[j] \textbf{ in} \\
\quad \textbf{else } msub(i,j,i) & \quad \textbf{if } k+1 = j \textbf{ then } s \\
 & \quad \textbf{else } \min(s, msub(i,j,k+1))
\end{array}
$$

Figure 1: Example programs.

These straightforward programs repeatedly solve common subproblems and take exponential time. For example, $m(i,j)$ computes $m(i,k)$ for all k from i to $j-1$ and computes $m(k,j)$ for all k from $i+1$ to j. We transform them into dynamic programming algorithms that perform efficient caching and take polynomial time.

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.

Caching requires appropriate data structures. In Step 1, we cache all possibly computed results in a recursive tree following the structure of recursive calls. Each node of the tree is a tuple that bundles recursive subtrees with the return value of the current call. We use $<>$ to denote a tuple, and we use selectors $1st$, $2nd$, $3rd$, etc. to select the first, second, third, etc. elements of a tuple.

In Step 2, cached values are used and maintained in efficiently computing function calls on slightly incremented inputs. We use an infix operation $\oplus$ to denote an input increment operation, also called an input change (or update) operation. It combines a previous input $x = \langle x_1, ..., x_n \rangle$ and an increment parameter $y = \langle y_1, ..., y_m \rangle$ to form an incremented input $x' = \langle x'_1, ..., x'_n \rangle = x \oplus y$, where each x'_i is some function of x_j's and y_k's. An input increment operation we use for program optimization always has a corresponding decrement operation $prev$ such that for all x, y, and x', if $x' = x \oplus y$ then $x = prev(x')$. Note that y might or might not be used. For example, an input increment operation to function m in Figure 1 could be $\langle x'_1, x'_2 \rangle = \langle x_1, x_2 + 1 \rangle$ or $\langle x'_1, x'_2 \rangle = \langle x_1 - 1, x_2 \rangle$, and the corresponding decrement operations are $\langle x_1, x_2 \rangle = \langle x'_1, x'_2 - 1 \rangle$ and $\langle x_1, x_2 \rangle = \langle x'_1 + 1, x'_2 \rangle$, respectively. An input increment to a function that takes a list could be $x' = cons(y, x)$, and the corresponding decrement operation is $x = cdr(x')$.

In Step 3, cached values that are not used for an incremental computation are pruned away, yielding functions that cache, use, and maintain only useful values. Finally, the resulting incremental program is used to form an optimized program. The optimized program computes in an incremental fashion with step $\oplus$, caching and reusing results of subcomputations as needed, and thus avoids repeatedly solving common subproblems.

For a function f in an original program, $\bar{f}$ denotes the function that caches all possibly computed values of f, and $\hat{f}$ denotes the pruned function that caches only useful values. We use x to denote an un-incremented input and use r, $\bar{r}$, and $\hat{r}$ to denote the return values of $f(x)$, $\bar{f}(x)$, and $\hat{f}(x)$, respectively. For any function f, we use f' to denote the incremental function that computes $f(x')$, where $x' = x \oplus y$, using cached results about x such as $f(x)$. So, f' may take parameter x', as well as extra parameters each corresponding to a cached result. Figure 2 summarizes the notation.

Function	Return Value	Denoted as	Incremental Function
f	original value	r	f'
$\bar{f}$	all possibly computed values	$\bar{r}$	$\bar{f}'$
$\hat{f}$	useful values	$\hat{r}$	$\hat{f}'$

Figure 2: Notation.

3 Step 1: Caching all possibly computed values

Consider a function f_0 defined by a set of recursive functions. Program f_0 may use global variables, such as x and y in function $c(i, j)$. A *possibly computed value* is the value of a function call that is computed for some but not necessarily all values of the global variables. For example, function $c(i, j)$ computes the value of $c(i-1, j-1)$ only when $x[i] = y[j]$. Such values occur exactly in branches of conditional expressions whose conditions depend on any global variable.

We construct a program $\bar{f}_0$ that caches all possibly computed values in f_0. For example, we extend $c(i, j)$ to always compute the value of $c(i-1, j-1)$ regardless of whether $x[i] = y[j]$. We first apply a simple *hoisting transformation* to lift function calls out of conditional expressions whose conditions depend on global variables. We then apply an *extension transformation* to cache all intermediate results, i.e., values of all function calls, in the return value.

3.1 Hoisting transformation

Hoisting transformation $\mathcal{H}st$ identifies conditional expressions whose condition depends on any global variable and then applies the transformation

$$\begin{array}{rl} \mathcal{H}st[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!] = & \textbf{let } v_2 = e_2 \textbf{ in} \\ & \textbf{let } v_3 = e_3 \textbf{ in} \\ & \textbf{if } e_1 \textbf{ then } v_2 \textbf{ else } v_3 \end{array}$$

For example, the hoisting transformation leaves m and $msub$ unchanged and transforms c into

$$
\begin{array}{l}
c(i,j) \triangleq \ \textbf{if } i = 0 \ \vee j = 0 \textbf{ then } 0 \\
\qquad\qquad \textbf{else let } u_1 = c(i-1, j-1) + 1 \textbf{ in} \\
\qquad\qquad\qquad \textbf{let } u_2 = \max(c(i, j-1), c(i-1, j)) \textbf{ in} \\
\qquad\qquad\qquad \textbf{if } x[i] = y[j] \textbf{ then } u_1 \textbf{ else } u_2
\end{array}
$$

$\mathcal{H}st$ simply lifts up the entire subexpressions in the two branches, not just the function calls in them. Administrative simplification performed at the end of the extension transformation will unwind bindings for computations that are used at most once in subsequent computations; thus computations other than function calls will be put down into the appropriate branches then. $\mathcal{H}st$ is simple and efficient. The resulting program has essentially the same size as the original program, so $\mathcal{H}st$ does not increase the running time of the extension transformation or the running times of the later incrementalization and pruning.

If we apply the hoisting transformation on arbitrary conditional expressions, the resulting program may run slower, become non-terminating, or have errors introduced, since the transformed program may perform certain computations not performed in some branches of the original program. By applying the hoisting transformation only on conditional expressions whose conditions depend on global variables, our assumption in Section 2 eliminates the last two problems. The first problem is discussed in Section 6.

3.2 Extension transformation

For each hoisted function definition $f(v_1, ..., v_n) \triangleq e$, we construct a function definition

$$\bar{f}(v_1, ..., v_n) \triangleq \mathcal{E}xt[\![e]\!]$$

where $\mathcal{E}xt[\![e]\!]$, defined in [38], extends an expression e to return a nested tuple that contains the values of all function calls made in computing e, i.e., it examines subexpressions of e in applicative order, introduces bindings that name the results of function calls, builds up tuples of these values together with the values of the original subexpressions, and passes these values from subcomputations to enclosing computations. The first component of a tuple corresponds to an original return value. Next, administrative simplifications clean up the resulting program. This yields a program $\bar{f}_0$ that embeds values of all possibly computed function calls in its return value. For the hoisted programs m and c, the extension transformation produces the following functions:

$$
\begin{array}{l}
\overline{m}(i,j) \triangleq \ \textbf{if } i = j \textbf{ then } < 0 > \\
\qquad\qquad \textbf{else } \overline{msub}(i,j,i) \\
\overline{msub}(i,j,k) \triangleq \ \textbf{let } v_1 = \overline{m}(i,k) \textbf{ in} \\
\qquad\qquad\qquad \textbf{let } v_2 = \overline{m}(k+1,j) \textbf{ in} \\
\qquad\qquad\qquad \textbf{let } s = 1st(v_1) + 1st(v_2) + p[i-1] * p[k] * p[j] \textbf{ in} \\
\qquad\qquad\qquad \textbf{if } k+1 = j \textbf{ then } < s, v_1, v_2 > \\
\qquad\qquad\qquad \textbf{else let } v = \overline{msub}(i,j,k+1) \textbf{ in} \\
\qquad\qquad\qquad\qquad < \min(s, 1st(v)), v_1, v_2, v > \\
\\
\bar{c}(i,j) \triangleq \ \textbf{if } i = 0 \ \vee j = 0 \textbf{ then } < 0 > \\
\qquad\qquad \textbf{else let } v_1 = \bar{c}(i-1, j-1) \textbf{ in} \\
\qquad\qquad\qquad \textbf{let } v_2 = \bar{c}(i, j-1) \textbf{ in} \\
\qquad\qquad\qquad \textbf{let } v_3 = \bar{c}(i-1, j) \textbf{ in} \\
\qquad\qquad\qquad \textbf{if } x[i] = y[j] \textbf{ then } < 1st(v_1) + 1, v_1, v_2, v_3 > \\
\qquad\qquad\qquad \textbf{else } < \max(1st(v_2), 1st(v_3)), v_1, v_2, v_3 >
\end{array}
$$

We have $m(i,j) = 1st(\overline{m}(i,j))$ and $c(i,j) = 1st(\overline{c}(i,j))$.

4 Step 2: Static incrementalization

The essence of our method is to transform a program to use and maintain cached values efficiently as the computation proceeds. This is done by incrementalizing $\bar{f}_0$ with respect to an input increment operation $\oplus$, i.e., we transform $\bar{f}_0(x \oplus y)$ to use the cached value of $\bar{f}_0(x)$ rather than compute from scratch.

An *input increment operation* $\oplus$ describes a minimal update to the input parameters. We first describe a general method for determining $\oplus$. We then give a method, called *static incrementalization*, that constructs an incremental version $\bar{f}'$ for each function $\bar{f}$ in the extended program and allows an incremental function to have multiple parameters that represent cached values.

4.1 Determining input increment operation

An input increment reflects how a computation proceeds. In general, a function may have multiple ways of proceeding, depending on the particular computations involved. There is no general method for identifying all of them or the most appropriate ones. Here we propose a method that can systematically identify a general class of them. The idea is to use a *minimal* input change that is in the *opposite* direction of change compared to arguments of recursive calls. Using the opposite direction of change yields an increment; using a minimal change allows maximum reuse, i.e., maximum incrementality.

Consider a recursively defined function f_0. Formulas for the possible arguments of recursive calls to f_0 in computing $f_0(x)$ can be determined statically. For example, for function $c(i,j)$, recursive calls to c have the set of possible arguments $S_c = \{\langle i-1, j-1\rangle, \langle i, j-1\rangle, \langle i-1, j\rangle\}$, and for function $m(i,j)$, recursive calls to m have the set of possible arguments $S_m = \{\langle i, k\rangle, \langle k+1, j\rangle \mid i \le k \le j-1\}$. The latter is simplified from $S_m = \{\langle a, c\rangle, \langle c+1, b\rangle \mid a \le c \le b-1, a = i, b = j\}$ where a, b, c are fresh variables that correspond to i, j, k in $msub$; the equalities are based on arguments of the function calls involved (in this case calls to $msub$); the inequalities are obtained from inequalities on these arguments (in the where-clause of $msub$). The simplification here, as well as the manipulations below, can be done automatically using Omega [50], a system for manipulating linear constraints over integer variables, Presburger formulas, and integer tuple relations and sets.

Represent the arguments of recursive calls so that the differences between them and x are explicit. For function c, S_c is already in this form, and for function m, S_m is rewritten as $\{\langle i, j - l\rangle, \langle i + l, j\rangle \mid 1 \le l \le j - i\}$. Then, extract minimal differences that cover all of these recursive calls. The partial ordering on differences is: a difference involving fewer parameters is smaller; a difference in one parameter with smaller magnitude is smaller; other differences are incomparable. A set of differences *covers* a recursive call if the argument to the call can be obtained by repeated application of the given differences. So, we first compute the set of minimal differences and then remove from it each element that is covered by the remaining elements. For function c, we obtain

$\{\langle i, j-1\rangle, \langle i-1, j\rangle\}$, and for function m, we obtain $\{\langle i, j-1\rangle, \langle i+1, j\rangle\}$. Elements of this set represent decrement operations *prev*. Finally, take the opposite of each decrement operation to obtain an increment operation $\oplus$, introducing a parameter y if needed (e.g., for increments that use data constructions). For function c, we obtain $\langle i, j+1\rangle$ and $\langle i+1, j\rangle$, and for function m, we obtain $\langle i, j+1\rangle$ and $\langle i-1, j\rangle$. Although finding input increment operations is theoretically hard in general (and a decrement operation might not have an inverse, in which case our algorithm does not apply), it is usually straightforward.

Typically, a function that involves repeatedly solving common subproblems contains multiple recursive calls to itself. If there are multiple input increment operations, then any one may be used to incrementalize the program and then form an optimized program; the rest may be used to further incrementalize the resulting optimized program, if it still involves repeatedly solving common subproblems. For example, for program c, either $\langle i, j+1\rangle$ or $\langle i+1, j\rangle$ leads to a final optimized program that takes polynomial time; the resulting program does not contain multiple recursive calls that solve common subproblems. For program m, either $\langle i-1, j\rangle$ or $\langle i, j+1\rangle$ leads to an optimized program that is an exponential factor faster, but the program still contains multiple recursive calls that solve common subproblems and takes exponential time; incrementalizing that program again under the other increment operation leads to a final optimized program that takes polynomial time. In other words, both $\langle i-1, j\rangle$ and $\langle i, j+1\rangle$ need to be used, and they may be used in either order.

4.2 Static incrementalization

Given a program $\bar{f}_0$ and an input increment operation $\oplus$, incrementalization symbolically transforms $\bar{f}_0(x')$ for $x' = x \oplus y$ to replace subcomputations with retrievals of their values from the value $\bar{r}$ of $\bar{f}_0(x)$. This exploits equality reasoning, based on control and data structures of the program and properties of primitive operations. The resulting program $\bar{f}_0'$ uses $\bar{r}$ or parts of $\bar{r}$ as additional arguments, called *cache arguments*, and satisfies: if $\bar{f}_0(x) = \bar{r}$ and $\bar{f}_0(x') = \bar{r}'$, then $\bar{f}_0'(x', \bar{r}) = \bar{r}'$.[1]

The idea is to establish the strongest invariants we can, especially those about cache arguments, for each function at all calls of it and maximize the usage of the invariants. At the end, unused candidate cache arguments are eliminated. Reducing running time corresponds to maximizing uses of invariants; reducing space corresponds to maintaining weakest invariants that suffice for all uses. It is important that the methods for establishing and using invariants are not only powerful but also systematic so that they are automatable. The algorithm is described below. Its use is illustrated afterwards using the running examples.

The algorithm starts with transforming $\bar{f}_0(x')$ for $x' = x \oplus y$ and $\bar{f}_0(x) = \bar{r}$ and first uses the decrement operation to establish an invariant about function arguments. More precisely, it starts with transforming $\bar{f}_0(x')$ with invariant $\bar{f}_0(prev(x')) = \bar{r}$, where $\bar{r}$ is a candidate cache argument. It may use other invariants about x' if given. Invariants given or formed from the enclosing conditions

[1] In previous papers, we defined $\bar{f}_0'$ slightly differently: if $\bar{f}_0(x) = \bar{r}$ and $\bar{f}_0(x \oplus y) = \bar{r}'$, then $\bar{f}_0'(x, y, \bar{r}) = \bar{r}'$. This difference is insubstantial since when incrementalization is used for program optimization, as we do here, both x and y (if it is used) can be obtained from x'.

and bindings are called *context*. The algorithm transforms function applications recursively. There are four cases at a function application $f(e'_1, ..., e'_n)$.

(i) If $f(e'_1, ..., e'_n)$ specializes, by definition of f, under its context to a base case, i.e., an expression with no recursive calls, then replace it with the specialized expression.

`Example.` For function application $f(e)$ with definition $f(x) \triangleq$ **if** $x \leq 0$ **then** 0 **else** $g(x)$ and context $e = 0$, we specialize $f(e)$ to 0.

(ii) Otherwise, if $f(e'_1, ..., e'_n)$ equals a retrieval from a cache argument based on an invariant about the cache argument that holds at $f(e'_1, ..., e'_n)$, then replace it with the retrieval.

`Example.` If invariant $f(e) = 2nd(\bar{r})$ holds at function application $f(e)$, then we replace $f(e)$ with $2nd(\bar{r})$.

(iii) Otherwise, if an incremental version f' of f has been introduced, then replace $f(e'_1, ..., e'_n)$ with a call to f' if the invariants associated with f' can be maintained; if some invariants cannot be maintained, then eliminate them and retransform from where f' was introduced. Maintaining invariants includes maintaining both the invariants about a cache argument, which have the form of a function application equaling a retrieval from a cache argument, and the other usual invariants.

`Example.` After introducing $\bar{f}_0'(x', \bar{r})$ to compute $\bar{f}_0(x')$ with invariant $\bar{f}_0(prev(x')) = \bar{r}$, we replace $\bar{f}_0(e)$ by $\bar{f}_0'(e, 3rd(\bar{r}))$ if we have $\bar{f}_0(prev(e)) = 3rd(\bar{r})$.

`Example.` After introducing $f'(x, r_1, r_2)$ to compute $f(x)$ with invariants $f(x-1) = r_1$, $f(x-2) = r_2$, and $x > 0$, if we encounter a function application $f(e)$ at which $f(e-1) = e_{r1}$ holds for some e_{r1} but neither $f(e-2) = e_{r2}$ for any e_{r2} nor $e > 0$ can be inferred, then we retransform from where f' was introduced and introduce $f'(x, r_1)$ with only invariant $f(x-1) = r_1$.

(iv) Otherwise, introduce an incremental version f' of f and replace $f(e'_1, ..., e'_n)$ with a call to f', as described below.

In general, the replacement in case (i) is also done, repeatedly, if the specialized expression contains only recursive calls whose arguments are closer to, and will equal after a bounded number of such replacements, arguments for base cases or arguments for which retrievals can be done. Since a bounded number of invariants are used at a function application, as described below, the retransformation in case (iii) happens at most a bounded number of times, so the algorithm always terminates; in the worst case, no invariants can be maintained and used, and f' is the same as f. Since f is just one function in the given program; the final program that uses f' after Step 3 might or might not be faster than the original program. There is no way to tell in general which is the case, but for the class of dynamic programming problems we consider, it can be conservatively determined that the original programs always contain repeated recursive calls and thus take exponential time, and the final programs proceed in nested linear fashion and thus take polynomial time.

Case (iv). To introduce an incremental version f' of f at $f(e'_1, ..., e'_n)$, we (iv.1) determine candidate invariants associated with f' based on the invariants that hold at $f(e'_1, ..., e'_n)$ and (iv.2) obtain a definition of f' based on the definition of f and the candidate invariants. Finally, we (iv.3) replace $f(e'_1, ..., e'_n)$ with a call to f'.

(iv.1) To determine candidate invariants associated with f', let *Inv* be the set of invariants about a cache argument or in the context that hold at $f(e'_1, ..., e'_n)$. Invariants about a cache argument are of the form $g_i(e_{i1}, ..., e_{in_i}) = e_{ri}$, where e_{ri} is either a candidate cache argument in the enclosing environment or a selector applied to such an argument. Invariants in the context are of a form given, e.g., $i \leq j$ for $m(i, j)$, or of the form $e = \mathit{true}$, $e = \mathit{false}$, or $v = e$ obtained from enclosing conditions or bindings. For simplicity, we assume that all bound variables are renamed so that they are distinct.

`Example.` As an example for Step (iv.1), consider function application $\overline{msub}(i', j', i')$, and assume that invariants about a cache argument, $\overline{msub}(i', j' - 1, i') = \bar{r}$ and $\overline{m}(i', j' - 1) = \bar{r}$, and invariants in the context, $i' \leq j'$, $i' \neq j'$, and $i' \neq j' - 1$, hold at the application.

We are introducing $f'(x''_1, ..., x''_n, ...)$ to compute $f(x''_1, ..., x''_n)$ for $x''_1 = e'_1, ..., x''_n = e'_n$, where $x''_1, ..., x''_n$ are fresh variables, and the second ... in the parameters of f' denote the cache arguments to be determined. So we deduce invariants about $x''_1, ..., x''_n$ based on *Inv* and $x''_1 = e'_1, ..., x''_n = e'_n$.

`Example.` For the example above, we let $\overline{msub}'$ compute $\overline{msub}(i'', j'', k'')$ for $i'' = i'$, $j'' = j'$, and $k'' = i'$, and deduce invariants about i'', j'', and k''.

The deduction has four steps.

(1) Use equations $e'_1 = x''_1, ..., e'_n = x''_n$ to try to eliminate all variables in *Inv* other than those in e_{ri}'s. This can be done automatically using Omega [50].

`Example.` For the example above, we give the following formula to Omega:

$$\exists i', j' : \quad \overline{msub}(i', j' - 1, i') = \bar{r}, \quad \overline{m}(i', j' - 1) = \bar{r}, \quad i' \leq j', \quad i' \neq j', \quad i' \neq j' - 1, \quad i' = i'', \quad j' = j'', \quad i' = k''$$

and we obtain the following result:

$$\overline{msub}(i'', j'' - 1, i'') = \bar{r}, \quad \overline{m}(i'', j'' - 1) = \bar{r}, \quad i'' \leq j'' - 2, \quad k'' = i''$$

(2) Remove resulting invariants that still use variables in *Inv* other than those in e_{ri}'s.

`Example.` For the example above, this has no effect.

(3) Use equations relating $x''_1, ..., x''_n$ to add additional forms of other invariants. This is done as follows: if $x''_j = x''_k$ or $x''_k = x''_j$ is a resulting equation, and i is another resulting invariant that involves x''_j, then for each invariant i' that can be obtained by replacing some occurrences of x''_j in i with x''_k, add i' to the resulting set of invariants.

`Example.` For the example above, this yields

$$\begin{array}{llll} \overline{msub}(i'', j'' - 1, i'') = \bar{r}, & \overline{msub}(i'', j'' - 1, k'') = \bar{r}, & \overline{m}(i'', j'' - 1) = \bar{r}, & i'' \leq j'' - 2, \quad k'' = i'', \\ \overline{msub}(k'', j'' - 1, i'') = \bar{r}, & \overline{msub}(k'', j'' - 1, k'') = \bar{r}, & \overline{m}(k'', j'' - 1) = \bar{r}, & k'' \leq j'' - 2, \end{array}$$

(4) For each invariant about a cache argument, replace its right side with a fresh variable.

`Example.` For the above example, six fresh variables, $\bar{r}_1$ to $\bar{r}_6$, are used to replace the right sides of the invariants about a cache argument.

We call the resulting invariants *candidate invariants*; each of them either uses only variables $x''_1, ..., x''_n$ or is of the form $g_i(e''_{i1}, ..., e''_{in_i}) = r_i$, where $e''_{i1}, ..., e''_{in_i}$ use only variables $x''_1, ..., x''_n$ and and r_i is a fresh variable. They are now associated with f', which has arguments $x''_1, ..., x''_n$ and candidate cache arguments r_i's.

Given invariants *Inv* and equations $x''_1 = e'_1, ..., x''_n = e'_n$, the set of strongest invariants about $x''_1, ..., x''_n$, expressed using no other variables in *Inv* except those in e_{ri}'s, are in general uncomputable. However, the deduction using Omega in (1) allows us to obtain such invariants automatically when only Presburger arithmetic is involved, which is the case for all the dynamic programming problems we consider. The removal in (2) allows us to fall back to weaker invariants in the general cases. The additional forms in (3) allow us to, when some invariants deduced can not be maintained at other calls to f, keep the strongest subset of invariants that can be obtained based on direct equalities.

(iv.2) To obtain a definition of f', first unfold $f(x''_1, ..., x''_n)$. Then exploit control structures, i.e., conditionals in $f(x''_1, ..., x''_n)$ and $g_i(e''_{i1}, ..., e''_{in_i})$'s, and data structures, i.e., components in r_i's, together with other candidate invariants associated with f'. Exploiting data structures allows us to use not only a cached result as a whole but also components of it. Exploiting control structures helps us obtain different forms of cached results under different conditions.

(1) To exploit conditionals in $f(x''_1, ..., x''_n)$, in the unfolded expression, move function applications into branches of the conditionals whenever possible, preserving control dependencies incurred by the order of conditional tests and data dependencies incurred by the bindings. This allows transformations of function applications to use as many conditions in their contexts as possible. This is done by repeatedly applying the following transformation in applicative, i.e., leftmost and innermost first, order to the unfolded expression:

For any expression $t(e_1, ..., e_k)$
being $c(e_1, ..., e_k)$, $p(e_1, ..., e_k)$, $f(e_1, ..., e_k)$, **if** e_1 **then** e_2 **else** e_3, or **let** $v = e_1$ **in** e_2:
if subexpression e_i is **if** e_{i1} **then** e_{i2} **else** e_{i3}
where if t is a conditional, $i \neq 2, 3$, and
if t is a binding expression, $i \neq 2$ or e_{i1} does not depend on v,
then transform $t(e_1, ..., e_k)$ to
if e_{i1} **then** $t(e_1, ..., e_{i-1}, e_{i2}, e_{i+1}, ..., e_k)$ **else** $t(e_1, ..., e_{i-1}, e_{i3}, e_{i+1}, ..., e_k)$.

`Example.` If the unfolded expression is **let** $v = h(e)$ **in if** e_1 **then** e_2 **else** e_3, where e_1 does not depend on v, then we transform it to **if** e_1 **then let** $v = h(e)$ **in** e_2 **else let** $v = h(e)$ **in** e_3.

This transformation preserves the semantics. It may increase the code size, but it does not increase the running time of the resulting program.

(2) To exploit the conditionals in $g_i(e''_{i1}, ..., e''_{in_i})$'s, first choose, among $g_i(e''_{i1}, ..., e''_{in_i})$'s that are applications of f, one whose arguments differ minimally from $x''_1, ..., x''_n$, denote it $f(e''_1, ..., e''_n)$, and call it the *corresponding previous application*. If the corresponding previous application is found, then introduce in the expression obtained from (1) conditions that appear in $f(e''_1, ..., e''_n)$, and put function applications inside both branches that follow such a condition. Again, this allows transformations of function applications to use as many conditions in their contexts as possible and, in particular, to use different forms of cached values from $f(e''_1, ..., e''_n)$ under different conditions. This is done by applying the following transformation in outermost-first order to the conditionals in the expression obtained from (1):

For each branch e of a conditional that contains a function application:
let e' be the condition of the leftmost and outermost conditional in $f(e''_1, ..., e''_n)$ such that the context of e does not imply e' and does not imply $\neg e'$;
if e' uses only variables defined in the context of e and takes constant time to compute, and the two branches in $f(e''_1, ..., e''_n)$ that are conditioned on e' contain different function applications in some component
then transform e to **if** e' **then** e **else** e.

`Example.` If a branch e of a conditional is $...h(x-1)...$, the corresponding previous application $f(x-1)$ by definition equals **if** $x-1=0$ **then** $<0>$ **else** $<e_1, e_2>$, and the context of e does not imply whether $x-1=0$ or not, then transform e to **if** $x-1=0$ **then** e **else** e.

Exploiting conditionals in the corresponding previous application $f(e''_1, ..., e''_n)$ is a heuristic. In general, one may exploit conditionals in all $g_i(e''_{i1}, ..., e''_{in_i})$'s; afterwards, conditionals whose two branches are the same are optimized by eliminating the condition and merging the two branches. We use this heuristic here since it simplifies the transformations and is sufficient for all examples we have seen. The rationale is that the arguments $e''_1, ..., e''_n$ differ minimally from $x''_1, ..., x''_n$, so the values of $f(e''_1, ..., e''_n)$ under various conditions are most likely to be reused in computing $f(x''_1, ..., x''_n)$.

(3) To exploit each component in a candidate cache argument r_i where there is an invariant $g_i(e''_{i1}, ..., e''_{in_i}) = r_i$, for each branch in the transformed expression from (2), specialize $g_i(e''_{i1}, ..., e''_{in_i})$ under the context of that branch. This may yield additional invariants that are equalities between function applications and components of r_i. It does not change the resulting expression from (2).

`Example.` Continuing the above example, if $f(x-1) = r$ is an invariant at e, and if e_2 is $h(x-1)$, then specializing $f(x-1)$ under $x-1 \neq 0$ yields $h(x-1) = 2nd(r)$. This invariant will enable one to replace $h(x-1)$ in e in the else-branch with $2nd(r)$.

After these control structures and data structures are exploited, we perform the following transformations on the expression from (2) in applicative order: we simplify subexpressions using algebraic properties and transform function applications recursively based on the four cases described.

(iv.3) After we finish transforming the expression for defining f', we eliminate dead code.

Finally, after we obtain a definition of f', replace the function application $f(e'_1, ..., e'_n)$ with a call to f' with arguments $e'_1, ..., e'_n$ and cache arguments e_{ir}'s for the invariants used.

After an application of f, other than the initial application $\bar{f}_0(x')$, is replaced by an application of f', if f' is not recursively defined, then we unfold the application of f' and repeat transformations (1) to (3) in (iv.2) on the enclosing expression that will become the body of the enclosing function. This enables, in defining the enclosing function, more exploitation of control structures and data structures based on the conditionals and binding expressions in the unfolded application of f'. This may increase the code size but not the running time of the resulting program.

4.3 Longest common subsequence

Incrementalize $\bar{c}$ under $\langle i', j' \rangle = \langle i+1, j \rangle$. We start with $\bar{c}(i', j')$, with cache argument $\bar{r}$ and invariant $\bar{c}(prev(i', j')) = \bar{c}(i'-1, j') = \bar{r}$; the invariants $i', j' > 0$ may also be included but do not affect any transformation below, so for brevity, we omit them. This is case (iv), so we introduce incremental version $\bar{c}'$ to compute $\bar{c}(i', j')$. Unfolding the definition of $\bar{c}$ and exploiting control structures according to (1) and (2) in (iv.2), we obtain the code below, where the annotations on the right are explained in the two paragraphs that follow. In particular, according to (2) in (iv.2), the false branch of $\bar{c}(i', j')$ is duplicated and put inside both branches of the additional condition $i'-1=0 \ \vee j'=0$, which is copied from the condition in the corresponding previous application $\bar{c}(i'-1, j')$; for convenience, the three function applications bound to v_1 through v_3 are not put inside branches that follow condition $x[i'] = y[j']$, since their transformations are not affected, and simplification at the end can take them back out.

$\bar{c}(i', j')$	with invariant $\bar{c}(i'-1, j') = \bar{r}$
=	
if $i'=0 \ \vee j'=0$ **then** $<0>$	
else if $i'-1=0 \ \vee j'=0$ **then**	context includes: $i'-1=0$
let $v_1 = \bar{c}(i'-1, j'-1)$ **in**	$=<0>$
let $v_2 = \bar{c}(i', j'-1)$ **in**	$= \bar{c}'(i', j'-1, \bar{c}(i'-1, j'-1)) = \bar{c}'(i', j'-1, <0>)$
let $v_3 = \bar{c}(i'-1, j')$ **in**	$=<0>$
if $x[i'] = y[j']$ **then** $< 1st(v_1)+1, v_1, v_2, v_3 >$	
else $< \max(1st(v_2), 1st(v_3)), v_1, v_2, v_3 >$	
else	context includes: $i' \neq 0, i'-1 \neq 0, j' \neq 0$
let $v_1 = \bar{c}(i'-1, j'-1)$ **in**	$= 3rd(\bar{r})$
let $v_2 = \bar{c}(i', j'-1)$ **in**	$= \bar{c}'(i', j'-1, \bar{c}(i'-1, j'-1)) = \bar{c}'(i', j'-1, 3rd(\bar{r}))$
let $v_3 = \bar{c}(i'-1, j')$ **in**	$= \bar{r}$
if $x[i'] = y[j']$ **then** $< 1st(v_1)+1, v_1, v_2, v_3 >$	
else $< \max(1st(v_2), 1st(v_3)), v_1, v_2, v_3 >$	

In the second branch (lines 2–7), $i'-1=0$ is true, since $j'=0$ would imply that the first branch is taken. Therefore, the first and third calls fall in case (i) and specialize to $<0>$. The second call falls in case (iii) and equals a recursive call to $\bar{c}'$ with arguments $i', j'-1$ and cache argument $<0>$, since we have a corresponding invariant $\bar{c}(i'-1, j'-1) = <0>$ from specialization. Additional simplification unwinds bindings for v_1 and v_3, simplifies $1st(<0>)+1$ to 1, and simplifies $\max(1st(v_2), 1st(<0>))$ to $1st(v_2)$.

In the third branch (lines 8–12), condition $i' - 1 = 0 \vee j' = 0$ is false; under this condition, the corresponding previous application $\bar{c}(i'-1, j')$ by definition of $\bar{c}$ equals its second branch where $\bar{c}(i'-1, j'-1)$ is bound to v_2, and thus the invariant $\bar{c}(i'-1, j') = \bar{r}$ implies $\bar{c}(i'-1, j'-1) = 3rd(\bar{r})$. Therefore, in this third branch, the first call falls in case (ii) and equals $3rd(\bar{r})$. The second call falls in case (iii) and equals a recursive call to $\bar{c}'$ with arguments $i', j'-1$ and cache argument $3rd(\bar{r})$ since we have a corresponding invariant $\bar{c}(i'-1, j'-1) = 3rd(\bar{r})$. The third call falls in case (ii) and equals $\bar{r}$. We obtain

$$
\begin{array}{l}
\bar{c}'(i', j', \bar{r}) \triangleq \ \textbf{if } i' = 0 \ \vee j' = 0 \textbf{ then } \ < 0 > \\
\qquad\qquad \textbf{else if } i' - 1 = 0 \textbf{ then} \\
\qquad\qquad\quad \textbf{let } v_2 = \bar{c}'(i', j' - 1, < 0 >) \textbf{ in} \\
\qquad\qquad\quad \textbf{if } x[i'] = y[j'] \textbf{ then } \ < 1, < 0 >, v_2, < 0 >> \\
\qquad\qquad\quad \textbf{else } \ < 1st(v_2), < 0 >, v_2, < 0 >> \\
\qquad\qquad \textbf{else let } v_1 = 3rd(\bar{r}) \textbf{ in} \\
\qquad\qquad\quad \textbf{let } v_2 = \bar{c}'(i', j' - 1, 3rd(\bar{r})) \textbf{ in} \\
\qquad\qquad\quad \textbf{let } v_3 = \bar{r} \textbf{ in} \\
\qquad\qquad\quad \textbf{if } x[i'] = y[j'] \textbf{ then } \ < 1st(v_1) + 1, v_1, v_2, v_3 > \\
\qquad\qquad\quad \textbf{else } \ < \max(1st(v_2), 1st(v_3)), v_1, v_2, v_3 >
\end{array}
$$

If $\bar{r} = \bar{c}(i'-1, j')$, then $\bar{c}'(i', j', \bar{r}) = \bar{c}(i', j')$, and $\bar{c}'$ takes time and space linear in j', for caching and maintaining a linear list. It is easy to see that $\bar{c}'$ takes linear time, since $\bar{c}'(i', j', ...)$ only calls $\bar{c}'(i', j'-1, ...)$ recursively, and $j' = 0$ is a base case. It is also easy to see that $\bar{c}'$ takes linear space, since each call to $\bar{c}'$ creates a tuple of length no more than 4.

4.4 Matrix-chain multiplication

Incrementalize $\overline{m}$ under $\langle i', j' \rangle = \langle i, j+1 \rangle$. We start with $\overline{m}(i', j')$, with cache argument $\bar{r}$ and invariants $\overline{m}(i', j'-1) = \bar{r}$ and $i' \leq j'$. This is case (iv), so we introduce incremental version $\overline{m}'$ to compute $\overline{m}(i', j')$. Unfolding $\overline{m}$, exploiting control structures according to (1) and (2) in (iv.2), and specializing the second branch according to case (i), we obtain the code below.

$$
\begin{array}{l}
\overline{m}(i', j') = \ \textbf{if } i' = j' \textbf{ then } \ < 0 > \\
\qquad\qquad\quad \textbf{else if } i' = j' - 1 \textbf{ then } \ < p[i'-1] * p[i'] * p[j'], < 0 >, < 0 >> \\
\qquad\qquad\quad \textbf{else } \overline{msub}(i', j', i')
\end{array}
\tag{1}
$$

In the third branch, condition $i' = j' - 1$ is false; under this condition, $\overline{m}(i', j'-1)$ by definition of $\overline{m}$ equals $\overline{msub}(i', j'-1, i')$, and thus the invariant $\overline{m}(i', j'-1) = \bar{r}$ implies $\overline{msub}(i', j'-1, i') = \bar{r}$. The call $\overline{msub}(i', j', i')$ falls in case (iv). We introduce $\overline{msub}'$ to compute $\overline{msub}(i'', j'', k'')$ for $i'' = i', j'' = j', k'' = i'$, with collected invariants

$$
\overline{msub}(i', j'-1, i') = \bar{r}, \quad \overline{m}(i', j'-1) = \bar{r}, \quad i' \leq j', \quad i' \neq j', \quad i' \neq j'-1 \tag{2}
$$

where the first two are about cache arguments; the third is given; and the last two are from the enclosing conditionals, in concise form, rather than, e.g., $(i' = j') = \mathit{false}$, for ease of reading. According to (iv.1), we express these invariants as invariants on i'', j'', k'' using Omega, and introduce fresh variables $\bar{r}_i$ for candidate cache arguments. We obtain candidate invariants associated with

$\overline{msub}'$:

$$\begin{array}{llll} \overline{msub}(i'', j''-1, k'') = \bar{r_1}, & \overline{m}(i'', j''-1) = \bar{r_2}, & i'' \leq j''-2, & k'' = i'', \\ \overline{msub}(i'', j''-1, i'') = \bar{r_3}, & & k'' \leq j''-2, & \\ \overline{msub}(k'', j''-1, k'') = \bar{r_4}, & \overline{m}(k'', j''-1) = \bar{r_5}, & & \\ \overline{msub}(k'', j''-1, i'') = \bar{r_6}, & & & \end{array} \quad (3)$$

Arguments of $\overline{msub}(i'', j''-1, k'')$ have a minimum difference from arguments of $\overline{msub}(i'', j'', k'')$, and thus $\overline{msub}(i'', j''-1, k'')$ is the corresponding previous application according to (2) in (iv.2).

Unfolding $\overline{msub}(i'', j'', k'')$ and exploiting control structures according to (1) and (2) in (iv.2), we obtain the code below, where the annotations on the right are explained in the four paragraphs that follow. In particular, according to (1) in (iv.2), the code for v_1 and v_2 is duplicated and moved into both branches that follow the condition $k''+1 = j''$. According to (2) in (iv.2), in the else-branch, the code for v is duplicated and put inside both branches that follow the additional condition $k''+1 = j''-1$, which is copied from the condition in the corresponding previous application $\overline{msub}(i'', j''-1, k'')$; for convenience, the code for v_1 and v_2 is not put inside branches that follow $k''+1 = j''-1$, since their transformations are not affected, and simplification at the end can take them back out.

```
msub'(i'', j'', k'')                                  start with invariants in (3), later with k'' = i'' eliminated
=
if k''+1 = j'' then
    let v1 = m'(i'', k'') in
    let v2 = m'(k''+1, j'') in
    let s = 1st(v1) + 1st(v2) + p[i''-1] * p[k''] * p[j''] in
    < s, v1, v2 >
else                                                  context includes: k''+1 ≠ j''
    let v1 = m'(i'', k'') in                          = <0> or 2nd(r1), where the former uses k'' = i''
    let v2 = m'(k''+1, j'') in                        = m'(k''+1, j'', m'(k''+1, j''-1)) = m'(k''+1, j'', 3rd(r1))
    let s = 1st(v1) + 1st(v2) + p[i''-1] * p[k''] * p[j''] in
    if k''+1 = j''-1 then                             context includes: k''+1 = j''-1
        let v = msub'(i'', j'', k''+1) in             = (4) below
        < min(s, 1st(v)), v1, v2, v >
    else                                              context includes: k''+1 ≠ j'', k''+1 ≠ j''-1
        let v = msub'(i'', j'', k''+1) in             = msub'(i'', j'', k''+1, msub'(i'', j''-1, k''+1), m'(i'', j''-1),
        < min(s, 1st(v)), v1, v2, v >                                  msub'(i'', j''-1, i''))
                                                      = msub'(i'', j'', k''+1, 4th(r1), r2, r3)
```

The first branch gets simplified away, since we have invariant $k'' \leq j''-2$.

In the else-branch, the corresponding previous application $\overline{msub}(i'', j''-1, k'')$ by definition of $\overline{msub}$ has $\overline{m}(i'', k'')$ bound to v_1 and $\overline{m}(k''+1, j''-1)$ bound to v_2, and thus the invariant $\overline{msub}(i'', j''-1, k'') = \bar{r_1}$ implies $\overline{m}(i'', k'') = 2nd(\bar{r_1})$ and $\overline{m}(k''+1, j''-1) = 3rd(\bar{r_1})$. The first call $\overline{m}(i'', k'')$ falls in case (i), since we have invariant $k'' = i''$, and equals $< 0 >$. The second call falls in case (iii) and equals a recursive call to $\overline{m}'$ with arguments $k''+1, j''$ and cache argument $3rd(\bar{r_1})$, since we have a corresponding invariant $\overline{m}(k''+1, j''-1) = 3rd(\bar{r_1})$.

In the branch where $k''+1 = j''-1$ is true, the call to $\overline{msub}$ falls in case (i) and equals

$$\textbf{let } v_1 = \overline{m}(i'', j''-1) \textbf{ in let } v_2 = \overline{m}(j'', j'') \textbf{ in let } s = 1st(v_1)+1st(v_2)+p[i''-1]*p[k''+1]*p[j''] \textbf{ in } < s, v_1, v_2 > \quad (4)$$

which then equals $<1st(\bar{r_2}) + p[i''-1] * p[k''+1] * p[j''],\ \bar{r_2}, <0>>$, because the first call equals $\bar{r_2}$, and the second call equals $< 0 >$.

In the last branch, the call to $\overline{msub}$ falls in case (iii). However, the arguments of this call do not satisfy the invariants corresponding to $k'' = i''$ or corresponding to those on the third and fourth lines of the candidate invariants in (3). Specifically, the invariant corresponding to $k'' = i''$ is $k''+1 = i''$, which is false because the context includes $k'' = i''$; the others can not be maintained because we can not infer that $\overline{msub}(k''+1, j''-1, k''+1)$, $\overline{m}(k''+1, j''-1)$, or $\overline{msub}(k''+1, j''-1, i'')$ equals a retrieval from any cache argument $\bar{r}_1$ to $\bar{r}_6$. So we delete these invariants and retransform $\overline{msub}$. Everything remains the same except that $\overline{m}(i'', k'')$ does not fall in case (i) any more; it falls in case (ii) and equals $2nd(\bar{r}_1)$. We replace this last call to $\overline{msub}$ by a recursive call to $\overline{msub}'$ with arguments $i'', j'', k''+1$ and cache arguments $4th(\bar{r}_1)$, $\bar{r}_2$, $\bar{r}_3$ since we have corresponding invariants $\overline{msub}(i'', j''-1, k''+1) = 4th(\bar{r}_1)$, $\overline{m}(i'', j''-1) = \bar{r}_2$, $\overline{msub}(i'', j''-1, i'') = \bar{r}_3$.

We eliminate unused candidate cache argument $\bar{r}_3$, and we replace the original call $\overline{msub}(i', j', i')$ in (1) with $\overline{msub}'(i', j', i', \bar{r}, \bar{r})$ according to (iv.3). We obtain

$$
\begin{array}{l}
\overline{m}'(i', j', \bar{r}) \triangleq \ \textbf{if } i' = j' \textbf{ then } < 0 > \\
\qquad\qquad \textbf{else if } i' = j' - 1 \textbf{ then } < p[i'-1] * p[i'] * p[j'], < 0 >, < 0 >> \\
\qquad\qquad \textbf{else } \overline{msub}'(i', j', i', \bar{r}, \bar{r}) \\
\\
\overline{msub}'(i'', j'', k'', \bar{r}_1, \bar{r}_2) \triangleq \\
\qquad\qquad \textbf{let } v_1 = 2nd(\bar{r}_1) \textbf{ in} \\
\qquad\qquad \textbf{let } v_2 = \overline{m}'(k''+1, j'', 3rd(\bar{r}_1)) \textbf{ in} \\
\qquad\qquad \textbf{let } s = 1st(v_1) + 1st(v_2) + p[i''-1] * p[k''] * p[j''] \textbf{ in} \\
\qquad\qquad \textbf{if } k''+1 = j''-1 \textbf{ then} \\
\qquad\qquad\qquad \textbf{let } v = < 1st(\bar{r}_2) + p[i''-1] * p[k''+1] * p[j''], \bar{r}_2, < 0 >> \textbf{ in} \\
\qquad\qquad\qquad < \min(s, 1st(v)), v_1, v_2, v > \\
\qquad\qquad \textbf{else let } v = \overline{msub}'(i'', j'', k''+1, 4th(\bar{r}_1), \bar{r}_2) \textbf{ in} \\
\qquad\qquad\qquad < \min(s, 1st(v)), v_1, v_2, v >
\end{array}
$$

Note that for the six invariants about cache arguments, $\bar{r}_1$ to $\bar{r}_6$, in (3), $\bar{r}_4$ to $\bar{r}_6$ can not be maintained at the recursive call and are weakened away; $\bar{r}_3$ can be maintained but is not used and hence is eliminated; and $\bar{r}_1$ and $\bar{r}_2$ can be maintained and are used and hence are kept.

For the resulting program $\overline{m}'$, if $\bar{r} = \overline{m}(i', j'-1)$, then $\overline{m}'(i', j', \bar{r}) = \overline{m}(i', j')$, and $\overline{m}'$ is an exponential factor faster. Function $\overline{m}'$ still takes exponential time due to repeated recursive calls to $\overline{m}'$, since each $\overline{m}'(i', j', ...)$ calls $\overline{m}'(k', j', ...)$ for all k from $i'+1$ to $j'-1$. Incrementalizing the resulting optimized program $\overline{m}(i, j)$ obtained from Step 3 under $\langle i', j' \rangle = \langle i-1, j \rangle$ yields a quadratic-time incremental program that involves no repeated recursive calls.

5 Step 3: Pruning unnecessary values

The first component of $\bar{f}_0'(x', \bar{r})$ is the return value of $f_0(x')$. Components of $\bar{r}$ that are not useful for computing this value need not be cached and maintained. We prune the programs $\bar{f}_0$ and $\bar{f}_0'$ to obtain a program $\hat{f}_0$ that caches only the useful values and a program $\hat{f}_0'$ that uses and maintains only the useful values. Finally, we form an optimized program that computes f_0 by using the base cases in $\hat{f}_0$ and by repeatedly using the incremental version $\hat{f}_0'$.

5.1 Pruning

Pruning requires a dependence analysis that can precisely describe substructures of trees [38]. We use an analysis based on regular tree grammars [31, 35]. We have designed and implemented a simple algorithm that uses regular tree grammar based constraints to efficiently produce precise analysis results [33, 35]. Pruning can save space and time and reduce code size.

For example, in program $\bar{c}'$, only the third component of $\bar{r}$ is useful. Pruning the second and fourth components of $\bar{c}$ and $\bar{c}'$, which makes the third component become the second component, and doing a few simplifications, which transform $1st(\bar{c})$ back to c and unwind bindings for v_1 and v_3, we obtain $\hat{c}$ and $\hat{c}'$ below:

$$\begin{array}{l}
\hat{c}(i,j) \triangleq \textbf{ if } i=0 \ \vee j=0 \textbf{ then } <0> \\
\qquad\qquad \textbf{else let } v_2 = \hat{c}(i,j-1) \textbf{ in} \\
\qquad\qquad\quad \textbf{if } x[i]=y[j] \textbf{ then } <c(i-1,j-1))+1, v_2> \\
\qquad\qquad\quad \textbf{else } <\max(1st(v_2), c(i-1,j))), v_2>
\end{array}$$

$$\begin{array}{l}
\hat{c}'(i',j',\hat{r}) \triangleq \textbf{ if } i'=0 \ \vee j'=0 \textbf{ then } <0> \\
\qquad\qquad\quad \textbf{else if } i'-1=0 \textbf{ then} \\
\qquad\qquad\qquad \textbf{let } v_2 = \hat{c}'(i',j'-1,<0>) \textbf{ in} \\
\qquad\qquad\qquad \textbf{if } x[i']=y[j'] \textbf{ then } <1, v_2> \\
\qquad\qquad\qquad \textbf{else } <1st(v_2), v_2> \\
\qquad\qquad\quad \textbf{else let } v_2 = \hat{c}'(i',j'-1,2nd(\hat{r})) \textbf{ in} \\
\qquad\qquad\qquad \textbf{if } x[i']=y[j'] \textbf{ then } <1st(2nd(\hat{r}))+1, v_2> \\
\qquad\qquad\qquad \textbf{else } <\max(1st(v_2), 1st(\hat{r})), v_2>
\end{array}$$

It is easy to see that $\hat{c}'$, like $\bar{c}'$, also takes time and space linear in j, but each call to $\hat{c}'$ creates a tuple of length no more than 2, compared to 4 for $\bar{c}'$.

Pruning leaves programs $\overline{m}$ and $\overline{m}'$ unchanged. We obtain the same programs $\widehat{m}$ and $\widehat{m}'$, respectively.

The first components of these functions remain unchanged, so we have $m(i,j) = 1st(\widehat{m}(i,j))$ and $c(i,j) = 1st(\widehat{c}(i,j))$.

5.2 Forming optimized programs

We redefine functions f_0 and $\hat{f}_0$ and use function $\hat{f}_0'$:

$$\begin{array}{l}
f_0(x) \triangleq 1st(\hat{f}_0(x)) \\
\hat{f}_0(x) \triangleq \textbf{ if } base_cond(x) \textbf{ then } base_val(x) \textbf{ else let } \hat{r} = \hat{f}_0(prev(x)) \textbf{ in } \hat{f}_0'(x,\hat{r})
\end{array}$$

where *base_cond* is the base-case condition, and *base_val* is the corresponding value, both copied from the original definition of $\hat{f}_0$ obtained by pruning. This new definition of $\hat{f}_0$ is called the optimized $\hat{f}_0$. In general, there may be multiple base cases, and we just list them all; to be conservative, we may include here all cases not containing multiple recursive calls.

For examples c and m, we obtain directly

$$\begin{array}{l}
c(i,j) \triangleq 1st(\hat{c}(i,j)) \\
\hat{c}(i,j) \triangleq \textbf{ if } i=0 \ \vee j=0 \textbf{ then } <0> \textbf{ else let } \hat{r} = \hat{c}(i-1,j) \textbf{ in } \hat{c}'(i,j,\hat{r})
\end{array}$$

$$\begin{array}{l}
m(i,j) \triangleq 1st(\widehat{m}(i,j)) \\
\widehat{m}(i,j) \triangleq \textbf{ if } i=j \textbf{ then } <0> \textbf{ else let } \hat{r} = \widehat{m}(i,j-1) \textbf{ in } \widehat{m}'(i,j,\hat{r})
\end{array}$$

It is easy to see that $\hat{c}(i,j)$ takes $O(i*j)$ time, since it only calls $\hat{c}(i-1,j)$ recursively, and $i=0$ is a base case; each call to $\hat{c}$ calls $\hat{c}'$, and $\hat{c}'$ takes $O(j)$ time. Thus, for $c(n,m)$, while the original program takes $O(2^{n+m})$ time, the optimized program takes $O(n*m)$ time. For $m(1,n)$, while the original program takes $O(n*3^n)$ time, the optimized program takes $O(n^2*2^n)$ time. Incrementalizing the optimized program again under the increment to the other parameter allows us to obtain a final optimized program that takes $O(n^3)$ time; the time complexity of the final optimized program is also easy to analyze.

The precise exponential complexities are not as easy to see, but for the purpose of our optimization, it is sufficient to know that they are exponential due to repeated recursive calls.

6 Summary and discussion

To summarize, we emphasize that it is the incremental computation under step $\oplus$ that determines appropriate values to cache so as to avoid repeated subcomputations. It yields a kind of regularity, in particular linearity, that we think is important for efficient computation.

Correctness. The transformations for caching, incrementalization, and pruning together preserve semantics in the sense that if the original $\hat{f}_0(x')$ terminates with a value, then the incremental program $\hat{f}_0'(x',\hat{r})$, given $\hat{r}=\hat{f}_0(prev(x'))$, terminates with the same value and is asymptotically at least as fast. This is because each transformation preserves semantics except that unfolding function definitions and eliminating unused values may make the resulting program terminate more often. Possible problems associated with hoisting causing $\hat{f}_0$ to compute values not computed in the original program f_0 are avoided as described in Section 3.1 and below. Forming optimized programs is straightforward for all the problems we have encountered, and it is easy to see that the resulting programs are correct, but a rigorous and general correctness argument for this needs further research. Overall, these transformations together preserve semantics in the sense that if the original program terminates with a value, then the optimized program terminates with the same value.

Mechanization. Our method for dynamic programming is composed completely of static program analyses and transformations and is systematic. It is based on a general approach for program optimization—incrementalization—which helps it to be systematic. The analyses and transformations used for caching and pruning are fully automatic and highly efficient [35, 38]. The analyses and transformations for incrementalization are fully automatic modulo the simplifications and equality reasoning used for establishing and using invariants. Such simplifications and equalities needed for all the problems we have encountered involve only Presburger arithmetic [50] and simple facts about recursive data structures and thus can be fully automated; for the same reason, determining input increment operations can also be fully automated. Also, as we have seen, forming optimized programs is straightforward to automate. Characterizing the exact class of problems to which these automated techniques apply needs further study.

Monovariance. Although our static incrementalization allows only one incremental version for each original function, i.e., is monovariant, it is still powerful enough to incrementalize all examples in [37, 38, 39], including various list manipulations, matrix computations, attribute evaluation, and graph problems. In general, while monovariant analyses and transformations are usually simpler and more efficient, they might not be sufficiently powerful when a function is used in multiple contexts for different roles. To overcome this potential problem, we propose to introduce a new function for each function composition that appears in the original program. This is based on the observation that different roles of a function are usually played in its composition with other functions. This method does not involve creating copies of existing functions, as is usually done but often causes code blowup and needs additional heuristics. We believe that the method based on static incrementalization can achieve dynamic programming for all problems whose solutions involve recursively solving subproblems that overlap, but a formal justification awaits more rigorous study.

Space usage and data structures. In our method, only values that are necessary for the incrementalization are stored, in appropriate data structures. For the longest-common-subsequence example, only a linear list is needed, whereas in standard textbooks, a quadratic two-dimensional array is used, and an additional optimization is needed to reduce it to a one-dimensional array [15]. For the matrix-chain multiplication example, our optimized program uses a list of lists that forms a triangle shape, rather than a two-dimensional array of square shape. It is nontrivial to see that recursive data structures give the same asymptotic speedup as arrays for some examples. Our recent work on transforming recursion into iteration can help eliminate the linear stack space used [36]. There are dynamic programming problems, e.g., 0-1 knapsack, for which the use of an array, with constant-time access to elements, helps achieve desired asymptotic speedups. Such situations become evident when doing incrementalization and can be accommodated easily, as will be described in a future paper. Although we present the optimizations for a functional language, the underlying principle is general and has been applied to programs that use loops and arrays [30, 34].

Auxiliary information. Some values computed in a hoisted program might not be computed by the original program and are therefore called *auxiliary information* [37]. Both incrementalization and pruning produce programs that are at least as fast as the given program, but caching auxiliary information may result in a slower program on certain inputs. We can determine statically whether such information is cached in the final program. If so, we can use time and space analysis [32, 54, 59, 60, 61] to determine conservatively whether it is worthwhile to use and maintain such information. The trade-off between time and space is an open problem for future study.

Additional properties. Many dynamic programming algorithms can be further improved by exploiting additional properties, such as greedy properties [7], of the given problems. For example, Hu and Shing [22, 23] give an $O(n * \log n)$-time algorithm for the matrix-chain multiplication

problem. Our method is not aimed at discovering such properties. Nevertheless, it can help preserve such properties once they are added. For example, for the paragraph-formatting problem [15, 18], we can derive a quadratic-time algorithm that uses dynamic programming; if the original program contains a simple extra conditional that follows from a kind of thinning property, our derived dynamic programming program uses it as well and takes linear time with a factor of line width. How to systematically discover and use such additional properties in general is a subject for future study.

7 Implementation and experimentation results

All three steps—caching, incrementalization, and pruning—have been implemented in a prototype system, CACHET, using the Synthesizer Generator [53]. Incrementalization as currently implemented is semi-automatic [29] and is being automated [63]. Determining input increment operations and forming optimized programs are currently done manually, but both are straightforward for all the problems we have encountered.

Figure 3 summarizes some of the examples derived, most of them semi-automatically and some automatically. The second column shows whether more than one cache argument is needed in an incremental program. The third column shows whether the incremental program computes values not necessarily computed by the original program. For the last two examples, the letter "a" in the third column shows that cached values are stored in arrays. The last two columns compare the asymptotic running times of the original programs and the optimized programs. For Fibonacci function, n is the input number, rather than the size of the input, and the running time is the numbers of additions performed. The matrix-chain multiplication, optimal binary search tree, and optimal polygon triangulation problems have similar control structures for recursive calls, which is reflected in the running times; yet, the optimal costs for these problems are computed in different ways, and our general method handles all of them in the same systematic manner. Paragraph formatting 2 [18] includes a conditional that reflects a greedy property, as described in Section 6.

Examples	multiple cache arg	aux info	original program's running time	optimized prog's running time
Fibonacci function [45]			$O(2^n)$	$O(n)$
binomial coefficients [45]			$O(2^n)$	$O(n*k)$
longest common subsequence [15]		√	$O(2^{n+m})$	$O(n*m)$
matrix-chain multiplication [15]	√		$O(n*3^n)$	$O(n^3)$
string editing distance [52]			$O(3^{n+m})$	$O(n*m)$
dag path sequence [6]		√	$O(2^n)$	$O(n^2)$
optimal polygon triangulation [15]	√		$O(n*3^n)$	$O(n^3)$
optimal binary search tree [2]	√		$O(n*3^n)$	$O(n^3)$
paragraph formatting [15]	√		$O(n*2^n)$	$O(n^2)$
paragraph formatting 2	√		$O(n*2^n)$	$O(n*width)$
0-1 knapsack [15]		√a	$O(2^n)$	$O(n*weight)$
context-free-grammar parsing [2]	√	√a	$O(n*(2*size+1)^n)$	$O(n^3*size)$

Figure 3: Summary of Examples.

To measure and compare the actual running times, we translated some of the programs into Java. The translation is straightforward, where tuples are implemented using class `Vector`. Other languages could be used, but Java is particularly good for testing whether caching additional information could incur a significant overhead in running time on small input, since operations on objects of the `Vector` class in Java are relatively expensive compared with operations on constructed data in languages such as ML, Scheme, or C.

Figure 4 presents the running times of the straightforward programs and the optimized programs for some examples; results for other examples are similar. Stars indicate running times longer than 48 hours. These measurements were taken for programs compiled with Sun JDK 1.0.2 and running on a Sun Ultra 10 with 300MHz UltraSPARC-IIi CPU and 128MB main memory. One can see that the optimized programs run much faster than the straightforward programs even on small inputs.

input	binomial coefficients		longest comm. subseq.		matrix-chain multipl.		paragraph formatting	
size	original	optim.	original	optim.	original	optim.	original	optim.
10	0	0	10	5	42	16	10	5
15	3	1	185	7	7453	23	111	7
20	305	2	36250	9	3282564	75	3712	10
25	4924	3	1454555	16	********	180	123360	16
40	2436874	6	********	60	********	752	********	68
60	********	11	********	113	********	2617	********	157
80	********	15	********	211	********	6187	********	325
100	********	20	********	383	********	11846	********	714

Figure 4: Running Times of Original Programs and Optimized Programs (in Milliseconds).

8 Related work and conclusion

Dynamic programming was first formulated by Bellman [4], where "programming" refers to the use of tabular solution method, and has been studied extensively since [58]. Bird [5], de Moor [17], and others have studied it in the context of program transformation. While some work addresses the derivation of recursive equations, including the original work by Bellman [4] and the later work by Smith [57], our work addresses the derivation of efficient programs that use tabulation. Previous methods for this problem either apply to specific subclasses of problems [11, 13, 14, 24, 46, 48] or give general frameworks or strategies rather than precise derivation algorithms [3, 5, 6, 8, 9, 16, 17, 45, 47, 55, 56, 62]. Our work is based on the general principle of incrementalization [37, 44] and consists of precise program analyses and transformations.

In particular, tupling [9, 46, 47] aims to compute multiple values together in an efficient way. It is improved to be automatic on subclasses of problems [11] and to work on more general forms [13]. It is also extended to store lists of values [48], but such lists are generated in a fixed way, which is not the most appropriate way for many programs. A special form of tupling can eliminate multiple data traversals for many functions [24]. A method specialized for introducing arrays was proposed for tabulation [12], but as our method has shown, arrays are not essential for the speedup

of many programs; their uses of arrays are complicated to derive and often take more space than necessary. For example, for longest common subsequence, binomial coefficients, string editing, dag path sequence, and 0-1 knapsack, quadratic space must be used there, while our derived programs requires only linear space, including the linear stack space [59], because of automatic garbage collection. To summarize, no previous method can perform all the powerful optimizations our method can. Each of our examples is non-trivial and requires advanced algorithm design discipline to derive even by hand.

Compared with our previous work for incrementalizing functional programs [37, 38, 39], this work contains several substantial improvements. First, our previous work addresses the systematic derivation of an incremental program f' given both program f and operation $\oplus$. This paper describes a systematic method for identifying an appropriate operation $\oplus$ given a function f and forming an optimized version of f using the derived incremental program f'. Second, since it is difficult to introduce appropriate cache arguments, our previous method allows at most one cache argument for each incremental function. This paper allows multiple cache arguments, without which many programs could not be incrementalized, e.g., the matrix-chain multiplication program. Third, our previous method introduces incremental functions using an on-line strategy, i.e., on-the-fly during the transformation, so it may attempt to introduce an unbounded number of new functions and thus not terminate. The algorithm in this paper introduces one incremental function for each function in the original program, i.e., it is monovariant; even though it is theoretically more limited, it is simpler, always terminates, and is able to incrementalize all previous examples. Finally, based on the idea of cache-and-prune [38], the method in this paper uses hoisting to extend the set of intermediate results [38] to include a kind of auxiliary information [37] that is sufficient for dynamic programming. This method is simpler than our previous general method for discovering auxiliary information [37]. Additionally, we now use a more precise and efficient dependence analysis for pruning [35].

Finite differencing [42, 43, 44] is based on the same underlying principle as incremental computation. Fifteen years ago, Paige explicitly asked whether finite differencing can be generalized to handle dynamic programming [42]; it is clear that he perceived an important connection. However, finite differencing has been formulated for set expressions in while loops [44], which can be obtained from fixed-point specifications [10], while straightforward solutions to dynamic programming problems are usually formulated as recursive functions, so it has been difficult to establish the exact connection. A major open problem is how to transform general recursive functions into set expressions extended with fixed-point operations [10].

Overall, being able to incrementalize complicated recursion in a general and systematic way is a substantial improvement complementing previous methods for incrementalizing loops [30, 44]. Our new method based on static incrementalization is both powerful and automatable. Based on our existing implementation, we believe that a complete system will perform incrementalization efficiently.

Acknowledgment

The authors would like to thank the anonymous referees for many helpful comments and suggestions.

References

[1] M. Abadi, B. Lampson, and J.-J. Lévy. Analysis and caching of dependencies. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 83–91. ACM, New York, 1996.

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

[3] F. L. Bauer, B. Möller, H. Partsch, and P. Pepper. Formal program construction by transformations—Computer-aided, intuition-guided programming. *IEEE Trans. Softw. Eng.*, 15(2):165–180, Feb. 1989.

[4] R. E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, New Jersey, 1957.

[5] R. S. Bird. Tabulation techniques for recursive programs. *ACM Comput. Surv.*, 12(4):403–417, Dec. 1980.

[6] R. S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Trans. Program. Lang. Syst.*, 6(4):487–504, Oct. 1984.

[7] R. S. Bird and O. de Moor. From dynamic programming to greedy algorithms. In B. Möller, H. Partsch, and S. Schuman, editors, *Formal Program Development*, volume 755 of *Lecture Notes in Computer Science*, pages 43–61. Springer-Verlag, Berlin, 1993.

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

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

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

[11] 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.

[12] W.-N. Chin and M. Hagiya. A bounds inference method for vector-based memoization. In ICFP 1997 [26], pages 176–187.

[13] 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.

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

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

[16] S. Curtis. Dynamic programming: A different perspective. In R. Bird and L. Meertens, editors, *Algorithmic Languages and Calculi*, pages 1–23. Chapman & Hall, London, U.K., 1997.

[17] 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.

[18] O. de Moor and J. Gibbons. Bridging the algorithm gap: A linear-time functional program for paragraph formatting. Technical Report CMS-TR-97-03, School of Computing and Mathematical Sciences, Oxford Brookes University, July 1997.

[19] J. Field and T. Teitelbaum. Incremental reduction in the lambda calculus. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 307–322. ACM, New York, 1990.

[20] 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.

[21] Y. Futamura and K. Nogi. Generalized partial evaluation. In B. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 133–151. North-Holland, Amsterdam, 1988.

[22] T. C. Hu and M. T. Shing. Computation of matrix chain products. part i. *SIAM J. Comput.*, 11(2):362–373, 1982.

[23] T. C. Hu and M. T. Shing. Computation of matrix chain products. part ii. *SIAM J. Comput.*, 13(2):228–251, 1984.

[24] Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling calculation eliminates multiple data traversals. In ICFP 1997 [26], pages 164–175.

[25] J. Hughes. Lazy memo-functions. In *Proceedings of the 2nd Conference on Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 129–146. Springer-Verlag, Berlin, Sept. 1985.

[26] *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*. ACM, New York, 1997.

[27] R. M. Keller and M. R. Sleep. Applicative caching. *ACM Trans. Program. Lang. Syst.*, 8(1):88–108, Jan. 1986.

[28] H. Khoshnevisan. Efficient memo-table management strategies. *Acta Informatica*, 28(1):43–81, 1990.

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

[30] Y. A. Liu. Principled strength reduction. In R. Bird and L. Meertens, editors, *Algorithmic Languages and Calculi*, pages 357–381. Chapman & Hall, London, U.K., 1997.

[31] Y. A. Liu. Dependence analysis for recursive data. In *Proceedings of the IEEE 1998 International Conference on Computer Languages*, pages 206–215. IEEE CS Press, Los Alamitos, Calif., 1998.

[32] Y. A. Liu and G. Gómez. Automatic accurate cost-bound analysis for high-level languages. *IEEE Transctions on Computers*, 50(12):1295–1309, Dec. 2001.

[33] Y. A. Liu, N. Li, and S. D. Stoller. Solving regular tree grammar based constraints. In *Proceedings of the 8th International Static Analysis Symposium*, volume 2126 of *Lecture Notes in Computer Science*, pages 213–233. Springer-Verlag, Berlin, 2001.

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

[35] Y. A. Liu and S. D. Stoller. Eliminating dead code on recursive data. In *Proceedings of the 6th International Static Analysis Symposium*, volume 1694 of *Lecture Notes in Computer Science*, pages 211–231. Springer-Verlag, Berlin, 1999.

[36] 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.

[37] 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, 1996.

[38] 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.

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

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

[41] D. J. Mostow and D. Cohen. Automating program speedup by deciding what to cache. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 165–172. Morgan Kaufmann Publishers, San Francisco, Calif., Aug. 1985.

[42] R. Paige. Programming with invariants. *IEEE Software*, 3(1):56–69, Jan. 1986.

[43] R. Paige. Symbolic finite differencing—Part I. In N. D. Jones, editor, *Proceedings of the 3rd European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 36–56. Springer-Verlag, Berlin, 1990.

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

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

[46] 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.

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

[48] 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.

[49] W. Pugh. An improved cache replacement strategy for function caching. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, pages 269–276. ACM, New York, 1988.

[50] W. Pugh. The Omega Test: A fast and practical integer programming algorithm for dependence analysis. *Commun. ACM*, 31(8):102–114, Aug. 1992.

[51] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 315–328. ACM, New York, 1989.

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

[53] T. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors.* Springer-Verlag, New York, 1988.

[54] M. Rosendahl. Automatic complexity analysis. In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture*, pages 144–156. ACM, New York, 1989.

[55] 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, 1981.

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

[57] 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.

[58] M. Sniedovich. *Dynamic Programming.* Marcel Dekker, New York, 1992.

[59] L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Automatic accurate stack space and heap space analysis for high-level languages. Technical Report TR 538, Computer Science Department, Indiana University, Apr. 2000.

[60] L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Automatic accurate live memory analysis for garbage-collected languages. In *Proceedings of the ACM SIGPLAN 2001 Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 102–111. ACM, New York, 2001.

[61] B. Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–538, Sept. 1975.

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

[63] Y. Zhang and Y. A. Liu. Automating derivation of incremental programs. In *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming*, page 350. ACM, New York, 1998.