Dynamic Programming via Static Incrementalization

Yanhong A. Liu* and Scott D. Stoller*

Abstract

Dynamic programming is an important algorithm design technique. It is used for solving problems whose solutions involve recursively solving subproblems that share subsubproblems. While a straightforward recursive program solves common subsubproblems repeatedly and often takes exponential time, a dynamic programming algorithm solves every subsubproblem just once, saves the result, reuses it when the subsubproblem is encountered again, and takes polynomial time. 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. The principle underlying incrementalization is general 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 and applied to numerous problems and succeeded on all of them.

1 Introduction

Dynamic programming is an important technique for designing efficient algorithms [2, 44, 13]. It is used for problems whose solutions involve recursively solving subproblems that overlap. While a straightforward recursive program solves common subproblems repeatedly, 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. This can reduce the time complexity from exponential to polynomial. The technique is generally applicable to all problems whose efficient solutions involve memoizing results of subproblems [4, 5].

Given a straightforward recursion, there are two traditional ways to achieve the effect of dynamic programming [13]: memoization [32] and tabulation [5].

Memoization uses a mechanism that is separate from the original program to save the result of each function call or reduction [32, 18, 21, 33, 23, 41, 43, 37, 17, 1]. 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 the table filling and lookup. Second, only values needed by the original program are actually computed, which is optimal in

*The authors gratefully acknowledge the support of NSF under Grant CCR-9711253. Authors' address: Computer Science Department, Indiana University, Lindley Hall 215, Bloomington, IN 47405. Phone: 812-855-{4373,7979}. Email: {liu,stoller}@cs.indiana.edu.

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 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, 12, 38, 37, 9, 11, 39, 40, 20, 10]. 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 generally requires a thorough understanding of the problem and a complete manual rewrite of the program [13]. 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 finally 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 are necessarily included.

Our method is based on static analyses and transformations studied previously by others [49, 8, 46, 6, 34, 19, 47, 39] and ourselves [31, 30, 29, 25, 30] and improves them. Yet, all three steps are simple, automatable, and efficient and have been implemented in a prototype system, CACHET. The system has been used to optimize many programs written as straightforward recursions, including all dynamic programming problems found in [2, 44, 13]. Performance measurements confirm drastic asymptotic speedups.

2 Formulating the problem

Straightforward solutions to many combinatorics and optimization problems can be written as simple recursions [44, 13]. For example, the matrix-chain-multiplication problem [13, 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 [13, 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 [13].

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
$\mid \quad c(e_1, ..., e_n)$	constructor application
$\mid \quad p(e_1, ..., e_n)$	primitive function application
$\mid \quad f(e_1, ..., e_n)$	function application
$\mid \quad$ **if** e_1 **then** e_2 **else** e_3	conditional expression
$\mid \quad$ **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 to be implicit parameters to functions; such variables can be identified easily for our language even if they are given as explicit parameters. 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. These examples do not use data constructors, but our previous papers contain a number of examples that use them [31, 30, 29] and our method handles them.

```
$c(i, j)$            where $i, j \geq 0$
$\triangleq$ if $i = 0 \ \vee j = 0$ then $0$
   else if $x[i] = y[j]$ then $c(i-1, j-1) + 1$
   else $\max(c(i, j-1), c(i-1, j))$
```

```
$m(i, j)$            where $i \leq j$
$\triangleq$ if $i = j$ then $0$
   else $msub(i, j, i)$

$msub(i, j, k)$            where $i \leq k \leq j - 1$
$\triangleq$ let $s = m(i, k) + m(k+1, j) + p[i-1] * p[k] * p[j]$ in
   if $k + 1 = j$ then $s$
   else $\min(s, msub(i, j, k+1))$
```

Figure 1: Example programs.

These straightforward programs repeatedly solve common subproblems and take exponential time. 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 $x = \mathit{prev}(x')$. Note that y need not be used. For example, an input increment 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 $x' = \mathit{cons}(y, x)$, and the corresponding decrement operation is $x = \mathit{cdr}(x')$.

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.

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 g, we use g' to denote the incremental function that computes $g(x')$, where $x' = x \oplus y$, using cached results about x such as $g(x)$. So, g' 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 $x[i] = y[j]$ but will be extended to always compute it. 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. 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.

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

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

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

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

$\mathcal{H}st$ 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 identify computations that occur at most once in subsequent computations and unwind the corresponding bindings; 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. For conditional expressions whose conditions depend on global variables, we assume that both branches may be executed to terminate correctly regardless of the condition, which holds for the large class of combinatorics and optimization problems we handle. By limiting the hoisting transformation on these conditional expressions, we eliminated the last two problems. The first problem is discussed in Section 6.

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

$$\bar{f}(v_1, ..., v_n) \triangleq \mathcal{E}xt[\![e]\!] \qquad (1)$$

where $\mathcal{E}xt[\![e]\!]$, defined in [30], 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:

```
$\overline{m}(i,j) \triangleq$ if $i = j$ then $<0>$
            else $\overline{msub}(i,j,i)$
$\overline{msub}(i,j,k) \triangleq$ let $v_1 = \overline{m}(i,k)$ in
                 let $v_2 = \overline{m}(k+1,j)$ in
                 let $s = 1st(v_1) + 1st(v_2) + p[i-1] * p[k] * p[j]$ in
                 if $k+1 = j$ then $< s, v_1, v_2 >$
                 else let $v = \overline{msub}(i,j,k+1)$ in
                        $< \min(s, 1st(v)), v_1, v_2, v >$

  $\bar{c}(i,j) \triangleq$ if $i = 0 \;\vee j = 0$ then $<0>$
            else let $v_1 = \bar{c}(i-1, j-1)$ in
                 let $v_2 = \bar{c}(i, j-1)$ in
                 let $v_3 = \bar{c}(i-1, j)$ in
                 if $x[i] = y[j]$ then $< 1st(v_1) + 1, v_1, v_2, v_3 >$
                 else $< \max(1st(v_2), 1st(v_3)), v_1, v_2, v_3 >$
```

4 Step 2: Static incrementalization

The essence of our method is to use and maintain cached values efficiently as a computation proceeds, i.e., we incrementalize $\bar{f}_0$ with respect to an input increment operation $\oplus$. Precisely, 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* $\oplus$ corresponds to a minimal update to the input parameters. We first describe a general method for identifying $\oplus$. We then give a powerful 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.

Input increment operation. An input increment should reflect 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 recursive calls involved (in this case *msub*); and the inequalities are obtained from the inequalities on these arguments. The simplification here, as well as the manipulations below, can be done automatically using Omega [42].

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. 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 $\langle i-1, j\rangle$. Even though finding an input increment operation is theoretically hard in general, it is usually straightforward.

Typically, a function involves repeatedly solving common subproblems when it contains multiple recursive calls to itself. If there are multiple input increment operations, then any one may be used to incrementalize the program and finally 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 $\langle i+1, j\rangle$ will lead to a final optimized program, and for program m, both $\langle i-1, j\rangle$ and $\langle i, j+1\rangle$ need to be used, and they may be used in either order.

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, especially those about cache arguments, at all calls and maximize their usage. At the end, unused candidate cache arguments are eliminated.

[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 $\bar{f}_0'(x, y, \bar{r}) = \bar{r}'$.

Reducing running time corresponds to maximizing uses of invariants; reducing space corresponds to maintaining weakest invariants for all uses. It is important that the methods for establishing and using invariants are specialized so that they are automatable. The precise 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 and bindings are called *context*. The algorithm transforms function applications recursively. There are four cases at a function application $f(e'_1, ..., e'_n)$.

(1) 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.

(2) Otherwise, if $f(e'_1, ..., e'_n)$ equals a retrieval from a cache argument based on an invariant about the cache argument in its context, then replace it with the retrieval.

(3) 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 corresponding invariants can be maintained; if some invariants can not be maintained, then eliminate them and retransform from where f' was introduced.

(4) 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 (1) 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 on which retrievals can be done. Since a bounded number of invariants are used at a function application, as described below, the retransformation in case (3) can only be done a bounded number of times. So, the algorithm always terminates.

To introduce an incremental version f' of f at $f(e'_1, ..., e'_n)$, let *Inv* be the set of invariants about cache arguments or context information at $f(e'_1, ..., e'_n)$. Those about cache arguments are of the form $g_i(e_{i1}, ..., e_{in_i}) = e_{ir}$, where e_{ir} is either a candidate cache argument in the enclosing environment or a selector applied to such an argument. Those about context information are of the form $e = true$, $e = false$, or $v = e$, obtained from conditions or bindings. For simplicity, we assume that all bound variables are renamed so that they are distinct. Introduce f' 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 deduce invariants about $x''_1, ..., x''_n$ based on *Inv*. The deduction uses equations $e'_1 = x''_1, ..., e'_n = x''_n$ to eliminate variables in *Inv* and can be done automatically using Omega [42]. Resulting equations relating $x''_1, ..., x''_n$ are used also to duplicate other invariants deduced. If a resulting invariant still uses a variable other than $x''_1, ..., x''_n$, discard it. Finally, for each invariant about a cache argument, replace its right hand side with a fresh variable, which becomes a candidate cache argument of f'. This yields the set of invariants now associated with f'. Note that invariants about cache arguments have 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 r_i is a fresh variable. Among the left hand sides of these invariants, identify an application of f whose arguments have a minimum difference from $x''_1, ..., x''_n$; if such an application exists, denote it $f(e''_1, ..., e''_n)$.

To obtain a definition of f', unfold $f(x''_1, ..., x''_n)$ and then exploit conditionals in $f(x''_1, ..., x''_n)$ and $f(e''_1, ..., e''_n)$ (if it exists) and components in the candidate cache arguments of f'. To exploit conditionals in $f(x''_1, ..., x''_n)$, move function applications inside branches of the conditionals in $f(x''_1, ..., x''_n)$

whenever possible, preserving control dependencies incurred by the order of conditional tests and data dependencies incurred by the bindings. This is done by repeatedly applying the following transformation in applicative order to the unfolded expression. For any $t(e_1,...,e_k)$ 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 e_i is **if** e_{i1} **then** e_{i2} **else** e_{i3}, where $i \neq 2,3$ if t is a conditional, and $i \neq 2$ or e_{i1} does not depend on v if t is a binding expression, 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** $t(e_1,...,e_{i-1},e_{i3},e_{i+1},...,e_k)$. This transformation preserves the semantics. It may increase the code size, but it does not increase the running time of the resulting program. To exploit the conditionals in $f(e''_1,...,e''_n)$, introduce conditions from $f(e''_1,...,e''_n)$ in the transformed expression just obtained and put function applications inside both branches that follow such a condition. This is done by applying the following transformation in outermost-first order to the conditionals in the transformed expression just obtained. For each branch e_i of the conditional that contains a function application, let e be the outermost condition in $f(e''_1,...,e''_n)$ that is not implied by the context of e_i; if e uses only variables defined in the context of e_i and takes constant time to compute, and the two branches in $f(e''_1,...,e''_n)$ that depend on e contain different function applications in some component, then transform e_i to **if** e **then** e_i **else** e_i. 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, specialize $g_i(e''_{i1},...,e''_{in_i})$ under the context of that branch. This may yield additional function applications that equal various components of r_i. After these control structures and data structures are exploited, we simplify primitive operations on $x'_1,...,x'_n$ and transform function applications recursively based on the four cases described. 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.

The simplifications and equality reasoning needed for all the problems we have encountered involve only recursive data structures and Presburger arithmetic and can be fully automated.

Longest common subsequence. Incrementalize 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 they are omitted for convenience. This is case (4), so we introduce incremental version $\bar{c}'$ to compute $\bar{c}(i',j')$. Unfolding the definition of $\bar{c}$ and listing conditions to be exploited, we obtain the code below. The false branch of $\bar{c}(i',j')$ is duplicated with the additional condition $i'-1=0 \ \vee j'=0$, which is copied from the condition in definition of $\bar{c}(i'-1,j')$; for convenience, three function applications bounded to v_1 to 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.

```
$\bar{c}(i',j') =$ if $i'=0 \ \vee j'=0$ then $<0>$
           else if $i'-1=0 \ \vee j'=0$ then
                let $v_1 = \bar{c}(i'-1,j'-1)$ in
                let $v_2 = \bar{c}(i',j'-1)$ in
                let $v_3 = \bar{c}(i'-1,j')$ in
                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 let $v_1 = \bar{c}(i'-1,j'-1)$ in
                let $v_2 = \bar{c}(i',j'-1)$ in
                let $v_3 = \bar{c}(i'-1,j')$ in
                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, $i'-1=0$ is true, since $j'=0$ would imply that the first branch is taken. The first and third calls fall in case (1) and specialize to $<0>$. The second call falls in case (3) and equals a recursive call to $\bar{c}'$ with arguments $i', j'-1$ and cache argument $<0>$ since we have a corresponding invariant $\bar{c}(i'-1, j'-1) = <0>$. 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, condition $i'-1=0 \vee j'=0$ is false; $\bar{c}(i'-1, j')$ by definition of $\bar{c}$ equals its second branch where $\bar{c}(i'-1, j'-1)$ is bound to v_2, and thus $\bar{c}(i'-1, j') = \bar{r}$ implies $\bar{c}(i'-1, j'-1) = 3rd(\bar{r})$. The first call falls in case (2) and equals $3rd(\bar{r})$. The second call falls in case (3) 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 (2) and equals $\bar{r}$. We obtain

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

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.

Matrix-chain multiplication. Incrementalize 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 $i' \le j'$. This is case (4), so we introduce incremental version $\overline{m}'$ to compute $\overline{m}(i', j')$. Unfolding $\overline{m}$, listing conditions, and specializing the second branch, we obtain the code below.

```
$\overline{m}(i', j') =$ if $i' = j'$ then $<0>$
            else if $i' = j' - 1$ then $<p[i'-1] * p[i'] * p[j'], <0>, <0>>$
            else $\overline{msub}(i', j', i')$
```

In the third branch, condition $i' = j'-1$ is false; $\overline{m}(i', j'-1)$ by definition of $\overline{m}$ equals $\overline{msub}(i', j'-1, i')$, and thus $\overline{m}(i', j'-1) = \bar{r}$ implies $\overline{msub}(i', j'-1, i') = \bar{r}$. The call $\overline{msub}(i', j', i')$ falls in case (4). We introduce $\overline{msub}'$ to compute $\overline{msub}(i'', j'', k'')$ for $i'' = i', j'' = j', k'' = i'$, with invariants $\overline{msub}(i', j'-1, i') = \bar{r}$, $\overline{m}(i', j'-1) = \bar{r}$, $i' \le j'$, $i' \ne j'$, $i' \ne j'-1$. Express these invariants as invariants on i'', j'', k'' using Omega, and introduce fresh variables $\bar{r_i}$ for candidate cache arguments. We obtain

$$\begin{array}{llllll} \overline{msub}(i'', j''-1, k'') = \bar{r_1}, & \overline{m}(i'', j''-1) = \bar{r_2}, & i'' \le j'', & i'' \ne j'', & i'' \ne j''-1, & k'' = i'', \\ \overline{msub}(i'', j''-1, i'') = \bar{r_3}, & & k'' \le j'', & k'' \ne j'', & k'' \ne j''-1, & \\ \overline{msub}(k'', j''-1, k'') = \bar{r_4}, & \overline{m}(k'', j''-1) = \bar{r_5}, & & & & \\ \overline{msub}(k'', j''-1, i'') = \bar{r_6}, & & & & & \end{array} \tag{2}$$

where equation $k'' = i''$ is an additional invariant deduced, and invariants not on the first line are duplications of those in the first line based on $k'' = i''$. Arguments of $\overline{msub}(i'', j''-1, k'')$ have a minimum difference from arguments of $\overline{msub}(i'', j'', k'')$.

Unfolding $\overline{msub}(i'', j'', k'')$ and listing conditions to be exploited, we obtain the following code. The code for v_1 and v_2 is duplicated for both branches that follow the condition $k''+1 = j''$. The code for v is duplicated for both branches that follow the additional condition $k''+1 = j''-1$, which is copied from the condition in the definition of $\overline{msub}(i'', j''-1, k'')$.

```
$\overline{msub}(i'',j'',k'') =$ if $k''+1=j''$ then
                     let $v_1 = \overline{m}(i'',k'')$ in
                     let $v_2 = \overline{m}(k''+1,j'')$ in
                     let $s = 1st(v_1)+1st(v_2)+p[i''-1]*p[k'']*p[j'']$ in
                     $< s, v_1, v_2 >$
                 else let $v_1 = \overline{m}(i'',k'')$ in
                     let $v_2 = \overline{m}(k''+1,j'')$ in
                     let $s = 1st(v_1)+1st(v_2)+p[i''-1]*p[k'']*p[j'']$ in
                     if $k''+1=j''-1$ then
                         let $v = \overline{msub}(i'',j'',k''+1)$ in
                         $< \min(s, 1st(v)), v_1, v_2, v >$
                     else let $v = \overline{msub}(i'',j'',k''+1)$ in
                         $< \min(s, 1st(v)), v_1, v_2, v >$
```

The first branch is simplified away since we have invariant $k'' \neq j''-1$.

In the other branch, $\overline{msub}(i'',j''-1,k'')$ 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 $\overline{msub}(i'',j''-1,k'') = \bar{r_1}$ implies $\overline{m}(i'',k'') = 2nd(\bar{r_1})$ and $\overline{m}(k''+1,j''-1) = 3rd(\bar{r_1})$. The first call falls in case (1), since we have invariant $k''=i''$, and equals $< 0 >$. The second call falls in case (3) 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 (1) and equals

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

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 (3). However, the arguments of this call do not satisfy the invariant corresponding to $k''=i''$ and those on the third and fourth lines in (2). So we delete these invariants and retransform $\overline{msub}$. Everything remains the same except that $\overline{m}(i'',k'')$ does not fall in case (1) any more; it falls in case (2) and equals $2nd(\bar{r_1})$. We replace this 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 $\overline{msub}(i'',j''-1,k''+1) = 4th(\bar{r_1})$, $\overline{m}(i'',j''-1) = \bar{r_2}$, $\overline{m}(i'',j''-1,i'') = \bar{r_3}$.

We eliminate unused candidate cache argument $\bar{r_3}$, and we replace the original call $\overline{msub}(i',j',i')$ by $\overline{msub}(i',j',i',\bar{r},\bar{r})$. We obtain

```
$\overline{m}'(i',j',\bar{r}) \triangleq$ if $i'=j'$ then $< 0 >$
               else if $i'=j'-1$ then $< p[i'-1]*p[i']*p[j'], < 0 >, < 0 >>$
               else $\overline{msub}'(i',j',i',\bar{r},\bar{r})$

$\overline{msub}'(i'',j'',k'',\bar{r_1},\bar{r_2}) \triangleq$ let $v_1 = 2nd(\bar{r_1})$ in
                        let $v_2 = \overline{m}'(k''+1,j'',3rd(\bar{r_1}))$ in
                        let $s = 1st(v_1)+1st(v_2)+p[i''-1]*p[k'']*p[j'']$ in
                        if $k''+1 = j''-1$ then
                            let $v = <1st(\bar{r_2})+p[i''-1]*p[k''+1]*p[j''],\ \bar{r_2}, < 0 >>$ in
                            $< \min(s, 1st(v)), v_1, v_2, v >$
                        else let $v = \overline{msub}'(i'',j'',k''+1,4th(\bar{r_1}),\bar{r_2})$ in
                            $< \min(s, 1st(v)), v_1, v_2, v >$
```

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. However, $\overline{m}$ still takes exponential time due to repeated calls to $\overline{m}'$; incrementalizing again under $\langle i',j'\rangle = \langle i-1,j\rangle$, we obtain a linear-time incremental program.

5 Step 3: Pruning unnecessary values

Among the components maintained by $\bar{f}_0'(x', \bar{r})$, the first one is the return value of $f_0(x')$. Components in $\bar{r}$ that are not useful for computing this value need not be cached and maintained. We prune the programs $\bar{f}_0$ and $\bar{f}_0'$ and 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 $\hat{f}_0'$.

Pruning. Pruning requires a dependence analysis that can precisely describe substructures of recursive trees [30]. We use an analysis method based on regular tree grammars [26]. We have implemented a simplified version that uses set constraints to efficiently produce precise analysis results. Pruning can save space, as well as 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 moves the third up to the second, and doing a few simplifications, which transforms $1st(\bar{c})$ back to c and unwinds bindings for v_1 and v_3, we obtain $\hat{c}$ and $\hat{c}'$ below:

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

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

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

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

$$f_0(x) \triangleq 1st(\hat{f}_0(x)) \quad \hat{f}_0(x) \triangleq \textbf{if } base\_cond(x) \textbf{ then } base\_val(x) \textbf{ else let } \hat{r} = \hat{f}_0(prev(x)) \textbf{ in } \hat{f}_0'(x, \hat{r})$$

where *base_cond* is the base-case condition, and *base_val* is the corresponding value, both copied from the definition of $\hat{f}_0$. In general, there may be multiple base cases, and we just list them all.

For examples c and m, we obtain directly

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

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

where $\hat{c}'$ and $\widehat{m}'$ are as obtained above. 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, we obtain an optimized program that takes $O(n^3)$ time.

6 Summary and discussion

Our method for dynamic programming is completely static, fully automatable, and efficient. In particular, it is based on a general approach for program optimization—incrementalization. Although our static incrementalization allows only one incremental version for each original function, it is still powerful enough to incrementalize all examples in [31, 30, 29], including various list manipulations, matrix computations, attribute evaluation, and graph problems. We believe that our method can perform dynamic programming for all problems whose solutions involve recursively solving subproblems that overlap, but a formal justification awaits more rigorous study.

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 [13]. 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's nontrivial to see that recursive data structures gives the same asymptotic speedup as arrays for these examples. There are dynamic programming problems, e.g., 0-1 knapsack, that require the use of array, with constant-time access of elements, to achieve the desired asymptotic speedup. Such need falls out explicitly when doing incrementalization and can be taken care of easily. This 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 [25, 28].

Some values computed in a hoisted program might not be computed by the original program and are therefore called *auxiliary information* [29]. Both incrementalization and pruning produce programs that are as 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 [27] to determine whether it is worthwhile to use and maintain such information.

Many dynamic programming algorithms can be further improved by exploiting greedy properties of the given problems [7]. Our method is not specially aimed at discovering such greedy properties. Nevertheless, it can maintain such properties once they are added. For example, for the paragraph-formatting problem [13, 16], we can derive a quadratic-time algorithm that uses dynamic programming; if the original program has a simple extra condition that follows from a greedy property, our derived dynamic programming program uses it as well and takes linear time with a factor of line width.

7 Implementation and experimentation results

All three steps have been implemented in a prototype system, CACHET. The incrementalization step is semi-automatic [24] and is currently being automated. The implementation uses the Synthesizer Generator [45].

Figure 3 summarizes some of the examples derived (most of them semi-automatically and some automatically) and compares their asymptotic running times. 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. Paragraph formatting 2 [16] includes a conditional that reflects a greedy property. The "a" in the third column for the last two examples shows that cached values are stored in arrays. Performance measurements confirmed drastic speedups.

Examples	multiple cache arg	aux info	original program's running time	optimized prog's running time
Fibonacci function [37]			$O(2^n)$	$O(n)$
binomial coefficients [37]			$O(2^n)$	$O(n*k)$
longest common subsequence [13]		√	$O(2^{n+m})$	$O(n*m)$
matrix-chain multiplication [13]	√		$O(n*3^n)$	$O(n^3)$
string editing distance [44]			$O(3^{n+m})$	$O(n*m)$
dag path sequence [6]		√	$O(2^n)$	$O(n^2)$
optimal polygon triangulation [13]	√		$O(n*3^n)$	$O(n^3)$
optimal binary search trees [2]	√		$O(n*3^n)$	$O(n^3)$
paragraph formatting [13]	√		$O(n*2^n)$	$O(n^2)$
paragraph formatting 2	√		$O(n*2^n)$	$O(n*width)$
0-1 knapsack [13]		√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.

8 Related work and conclusion

Dynamic programming was first formulated by Bellman [4] and has been studied extensively since [48]. Bird [5], de Moor [15], and others have studied it in the context of program transformation. These previous methods either apply to specific subclasses of problems [12, 38, 9, 11, 40, 20] or give general frameworks and strategies rather than precise algorithms [49, 8, 5, 46, 6, 3, 37, 47, 15, 39, 14]. Our work is based on the general principle of incrementalization [36, 29] and consists of precise analyses and transformations.

In particular, tupling [38, 39] aims to compute multiple values together in an efficient way. It is improved to be automatic on subclasses of problems [9] and to work on more general forms [11]. It is also extended to store lists of values [40], 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 [20]. A method specialized for introducing arrays was proposed for tabulation [10], but as our method has shown, array is not essential for the speedup of many programs; their arrays are complicated to derive and often consume more space than necessary.

Compared with our previous work for incrementalizing functional programs [31, 30, 29], this work contains drastic improvements. First, our previous work address 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 using the derived incremental program f' to form an optimized version of 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 statically determines one incremental function for each one 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 that was proposed earlier [30], the method in this paper uses hoisting to extend the set of intermediate results [30] to include a kind of auxiliary information [29] that is sufficient for dynamic programming. This method is simpler than our previous general method for discovering auxiliary information [29]. Additionally, we now use a more precise and efficient dependence analysis for pruning [26].

Finite differencing [36, 35] is based on the same underlying principle as incremental computation. Paige has explicitly asked whether finite differencing can be generalized to handle dynamic programming [34]; it is clear that he perceived an important connection. However, finite differencing has been formulated for set-based languages, while straightforward solutions to dynamic programming problems are usually formulated as recursive functions, so it was difficult to actually establish the connection.

Overall, being able to incrementalize complicated recursion in a systematic way is a more drastic improvement complementing previous methods for incrementalizing loops [36, 25]. Our new method based on static incrementalization is general and fully automatable. Based on our existing implementation, we believe that a complete system will perform incrementalization efficiently.

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

[9] 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, June 1993.

[10] W.-N. Chin and M. Hagiya. A bounds inference method for vector-based memoization. In ICFP 1997 [22], pages 176–187.

[11] 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.

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

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

[14] 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.

[15] 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.

[16] 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.

[17] 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, June 1990.

[18] 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.

[19] 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.

[20] Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling calculation eliminates multiple data traversals. In ICFP 1997 [22], pages 164–175.

[21] 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.

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

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

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

[25] 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.

[26] 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., May 1998.

[27] Y. A. Liu and G. Gómez. Automatic accurate time-bound analysis for high-level languages. In *Proceedings of the ACM SIGPLAN 1998 Workshop on Languages, Compilers, and Tools for Embedded Systems*, volume 1474 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1998.

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

[29] 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.

[30] 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.

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

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

[33] 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.

[34] R. Paige. Programming with invariants. *IEEE Software*, pages 56–69, Jan. 1986.

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

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

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

[38] 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.

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

[40] 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.

[41] 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, July 1988.

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

[43] 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, Jan. 1989.

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

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

[46] 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.

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

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

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