EFFICIENT HANDLING OF DATA STRUCTURES IN DEFINITIONAL LANGUAGES

Boleslaw K. SZYMANSKI
Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY 12180-3590, U.S.A.

Noah S. PRYWES
Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, U.S.A.

Communicated by A. Pnueli
Received August 1983
Revised January 1987

Abstract. Implementations of operations on general data structures in definitional languages often lead to excessive copying and storage requirements. To partially overcome this problem, users are given facilities to select efficient storage structures or to guide storage allocation. This contradicts the spirit of definitional languages, requiring the user to get involved with implementation details.

This paper presents a method for automatically recognizing excessive copying and optimizing the storage for data structures. Based on analysis of data dependencies, the storage may be reduced from an entire structure to individual elements of the structure. The benefits are especially significant in incremental structures, where only a constant number of elements of a large data structure is modified in each operation. For incremental structures, copying of unchanged parts of the structure is avoided, and unnecessary iterations are eliminated, without involving the user in this consideration. The user is thus relieved of considering the inefficiencies inherent in specifications in definitional languages. The method is applicable to a variety of language processors and computer architectures. The proposed optimization method produces better results than those obtained by explicit storage references.

The paper describes the implementation of the method in the compiler of the MODEL definitional language. First, criteria are presented for recognizing structures that may be optimized. Then, a transformation that removes iterations implied by the operations on incremental structures is described. The method is exemplified by its application to the non-recursive iteration computation of the Ackermann's function.

1. Introduction

Definitional languages, as opposed to procedural ones, interpret assignments as mathematical equations, i.e. the two sides of the assignment are declared to be equal. To assure that, these languages follow the single assignment rule—each variable has exactly one value. Thus programs in definitional languages are free from side-effects, and more amenable to program verification (including formal verification methods) and to parallel processing. To distinguish definitional programs
from the procedural programs we refer to the former as specifications. Some definitional languages, like LUCID [8] or FP [9], were designed primarily with verification in mind. Other declarative languages are motivated by the lack of explicit sequential control, inherent in the languages, which makes them well-suited for concurrent programming [12] and programming of data-flow machines [1]. A description of the mechanism of data-flow machines may be found in [2], [14], [26] and [35]. Yet other definitional languages were designed for programming by equations [25] and [12]. Some classes of definitional languages are variously called descriptive, nonprocedural (e.g. LUCID [8], MODEL [36] or iterative (MEDEE [17] Hiobol [38] and Xloop [41]. While interpretation of specifications usually uses lazy evaluation [22], compilation typically translates specifications into a group of iterations over topologically sorted equations [15, 31].

In exchange for the semantic elegance of specifications in definitional languages, their straightforward execution tends to require excessive run-time storage and unnecessary copying (partial or complete) of data structures. Such overhead mainly results from the single value restriction that requires the definition of a new complete data structure even when only a small part of it has to be changed. A typical example of unnecessary copying is the implementation of stack operations. A stack is usually defined as an array (or list) of elements together with a pointer (or index) to the top element. Each change on the top of a stack is described in a definitional language by defining an entirely new array, and involves copying of the unchanged elements of the old array. A more efficient, procedural implementation could modify only the last element, which can be changed, added or dropped. Another example of such inefficiency is an implementation of queues. In general that problem arises for any structure that changes only a small part of its value in each operation. A structure will be called incremental, if the number of its elements affected by each operation is constant.

Several techniques have been proposed for efficient handling of operations on data structures in definitional languages. Most LISP implementations, for example, allow explicit "store" operations giving the users more control over management of storage. Such a feature, however, destroys the definitional properties of the language, and several advantages of definitional languages are therefore lost. Similarly, in the general case of functional languages, guidance commands have been proposed as a tool for efficient handling of arrays [21]. The user is allowed to specify via the SAVE command how many recent elements of an array should be stored. The disadvantage is again the changed semantics of the specification and difficulty in checking correctness of use of guidance commands.

One method of efficient handling of operations on data structures in definitional languages is to use specialized structures. The classical example of such special structures, that are implemented by sharing rather than copying, is the list as implemented in pure LISP. More recent examples are streams. A stream is a list constructed using a non-strict cons operator. Therefore, elements of a stream can be accessed before the entire stream is created. If a stream is constructed and
accessed sequentially, only the part of it that is already created but not yet accessed has to be retained in memory. Another technique of reducing storage overhead while retaining definitional semantics of a language is variously called lenient cons [18] or I-structures [7]. An I-structure is a data structure with monotonical construction and access. Efficient implementation schemes for I-structures with known bounds have been proposed. However, correctness of the use of I-structures cannot be checked until run-time. Furthermore, the order of nesting and scope of loops connected with operations on I-structures follows the user's specification, regardless of efficiency. Both streams and I-structures have to be constructed and accessed sequentially. Therefore, a sequence of operations decreasing and increasing the size of a structure cannot be implemented by I-structures or streams.

The approach presented in this paper is based on the premise that schemes for efficient operations on data structures should be devised automatically. Users should be concerned only with the correctness of specifications without regard to efficiency and the system's task is to map the user's structures into efficient memory representations. Automatic recognition and efficient handling of data structures require an analysis of the data structure declarations and references. This analysis is easier to conduct in a compiler, before execution, than dynamically in an interpreter. The discussion in the paper, although valid for any definitional language scheme of execution, is illustrated by the optimization in the compiler (program generator) for the MODEL language [36]. No prior knowledge of MODEL is required to follow the presentation of this paper.

The MODEL language and its compiler into PL/I have been used for mathematical modeling and business data processing. The language is nonprocedural and the compiler automatically handles global analysis and consistency checking of specifications. Using different code generators, the compiler can produce code for dataflow machines [19] and distributed [39], concurrent [37] and real-time systems [40, 29]. High-level operations on aggregates of data are permitted; for optimization purposes the MODEL system decomposes and translates them into elemental operations [30]. The MODEL language allows declaration and use of hierarchical data structures with repetitive nodes. Such structures are capable of modeling other data types used by definitional languages such as lists, streams, arrays and structures with selectors. Thus, the data analysis method designed for the MODEL language is representative and transferable to other definitional languages.

It should be noted that because the object program is produced in a high-level programming language (for MODEL in C, FORTRAN or PL/I), it is further subjected to classical optimization methods incorporated in the compiler of the object language [3]. However, definitional languages require an additional level of optimization. We should realize that in procedural languages optimization can be regarded as a luxury that on the average can improve efficiency by a percentage. In definitional languages it is a necessity and the potential gain can be by an order of magnitude. The method of optimization presented in this paper can provide such gain for large incremental structures.
The optimization is achieved by motion of code out of loops and the removal of empty loops generated by such movements. Statements satisfying conditions for code motion are generated by the compiler from equations that describe operations on incremental structures. Such operations are often used in nonprocedural specifications (e.g. stacks that are used for iterative description of recursive algorithms [5]). Detection of statements that can be moved is based on the analysis of conditions in conditional statements. We are not aware of any existing optimizer of high-level language that performs such deep analysis.

The paper is organized as follows. In Section 2, data structures that have been used or proposed for definitional languages are reviewed and compared with MODEL hierarchical data structures. Storage representation of data structures in definitional languages is discussed in Section 3. This section also explains how the MODEL system transforms specifications to create efficient storage representations of data structures in the generated object programs. In Section 4, the method of efficient handling of memory and operations on incremental data structures is described. First, the criteria for recognizing structures that can be optimized are introduced. Then the transformation that removes excessive iterations connected with operations on incremental structures is described. The method is illustrated through an example of non-recursive iterative computation of Ackermann's function [32] in Section 5. First, we present the user's specification using operations on high-level structures and its source-to-source translation to elementary operations. Next, the schedule of program events generated by the system is described. Then, optimization of storage allocated for data structures is discussed. Finally, a more efficient transformed program for computation of Ackermann's function is presented. Section 6 concludes the paper and describes the result of implementation of the presented method in the MODEL system.

2. Data structures in definitional languages

Several data types have been proposed for use in definitional languages. The best known among them, introduced in LISP, is the list structure and the operations associated with it. A list is either the distinguished (empty) list [ ] or a pair (value, list). A value is any element of the respective domain of the list, including another list. Traditionally, the following three operations are defined on lists:

- **car(s)** — that returns the first element of the list s.
- **cdr(s)** — that returns the second element (i.e. the tail) of the list s.
- **cons(v, s)** — that creates a list whose first element is v and the tail is s.

If the argument list s is empty (i.e. equal to [ ]) then **car(s)** and **cdr(s)** return an undefined value τ. We assume that the operation **cons** is strict for lists, i.e. if any
argument of \texttt{cons} is the undefined value \(\tau\) then the result is also the undefined value \(\tau\). Dropping this assumption leads to a different data type called \textit{stream}. A stream is a list constructed using a nonstrict \texttt{cons} operator. Streams have proved to be useful in languages using lazy evaluation.

The most general data type used in definitional languages is a structure with selectors [13] which can represent both streams and lists. A structure is either the distinguished empty structure (\()\) or a set of (index, value) pairs such that all indexes in the set are unique. An index is an integer. A value represents a structure or elemental value (e.g. an integer or real number, character or bit string). Two basic operations on structures, \texttt{select} and \texttt{insert}, are defined as follows:

\texttt{select} \((s, i)\), denoted also as \(s(i)\), returns a value \(v\) such that \((i, v) \in s\) if \(i\) is an index in \(s\), or the empty structure (\()), otherwise.

\texttt{insert} \((s, i, v)\) creates the empty structure (\())\) if \(v = (\())\), or a structure \(s'\) such that \(s'(j) = s(j)\) for \(j \neq i\) and \(s'(i) = v\), otherwise.

Structures with selectors are used for example in Id [6].

The basic data structure in MODEL is a hierarchical multi-dimensioned structure which is an extension of the PL/I data structure. The primitive data types are similar to those allowed in PL/I: Picture, Decimal (Fixed or Float), Binary, Bit and Character. They are declared by a declaration of the form:

\[ X \langle\text{dimension}\rangle \text{ is field } \langle\text{description}\rangle, \]

where \langle\text{dimension}\rangle is an optional dimension specification of the form \((d_1, \ldots, d_n), n > 0\); and \langle\text{description}\rangle is a PL/I primitive type specification, such as NUM(4). The dimensions \(d_1, \ldots, d_n\) may be integer constants, or the special symbol *, designating a variable size dimension. For a variable size dimension, it is expected that some declaration in the program, such as \textit{SIZE} \(X = N\), defines the actual size. Alternately, the size of \(X\) may be inferred from an equation such as \(X(J) = Y(J) + 1\) that implicitly relates the size of \(X\) to that of \(Y\) which may be explicitly declared.

A compound (non-primitive) data structure is declared by a definition of the form:

\[ lv \ Y \langle\text{dimension}\rangle \text{ is group}, \]

\[ lv + 1 \quad \text{DCL}_1, \]
\[ \vdots \]
\[ lv + 1 \quad \text{DCL}_n, \]

where \text{DCL}_1, \ldots, \text{DCL}_n are declarations of other primitive or compound structures that are considered to be the parts of the structure \(Y\). As before, \langle\text{dimension}\rangle is an optional dimension. MODEL uses the PL/I (COBOL) level indicators \(lv, lv + 1, \ldots, lv + 1\) that denote that \(Y\) is a structure of level \(lv\), containing the substructures of level \(lv + 1\) declared by the declarations \text{DCL}_1, \ldots, \text{DCL}_n. Consider, for example,
the following declaration:

1  A(5) IS GROUP,
2  B IS FIELD (PIC(99)),
2  C(*) IS GROUP,
3  D IS FIELD (NUM(6)).

Note that D is nested within two dimensioned structures. Therefore, references to it should include two subscripts. The reference may be in the form D(I, J) (if no other structure in the program is named D), or also A(I).C(J).D, C(I, J).D, etc.

If the structure represents information that has to be transmitted from or to external storage, it is required that the complete structure (level 1) be declared a FILE, rather than a GROUP. In this case, it is also required that one intermediate level between FILE and FIELD be declared as RECORD. This specifies to the system the substructure that can be transferred from or to external storage in one read or write operation.

Thanks to the possibility of having a run-time determined dimension, provided by the * dimension designator, it is possible to represent structures such as lists or stacks in the basic MODEL data structure.

Figure 2 illustrates a complete declaration of the data structures that are used in the example to be later studied.

To operate on structures with indexes, an iteration block was introduced in several definitional languages. In some of them (e.g. Id [6], VAL [33]) the iteration block is explicitly defined and its main feature is a loop variable that may assume several values. The iteration block therefore consists of:

1. The initial value of the loop variable.
2. A test to determine the loop termination condition.
3. An expression to compute the next value for the loop variable or the value to be returned, if the loop terminates. This expression typically depends on the current value of the loop variable.

Such a solution however forces the user to define an order of nesting and scopes of the loops. The entire structure produced in one loop has to be transferred to the other loops in which it is used. Therefore, the disadvantage of this solution is that it requires storing entire structures in the main memory and decreases parallelism in loop execution (use of a structure cannot start before its construction is completed).

In the MODEL language, iterations are expressed implicitly by use of a special type of variable called *subscripts*. A subscript can assume any integer value from 1 to the size of an array dimension associated with that subscript. Thus, in the MODEL language, each equation with subscripts defines and refers to multidimensional structures. The size of each dimension may be separately given by a data declaration, a special statement defining a variable dimension, or may be implied by the sizes of other dimensions.

In summary, the MODEL hierarchical data structures, trees with nodes representing repeating substructures, are capable of representing data types used in definitional languages. Iterations in MODEL are expressed implicitly leaving room for optimal arrangements of their scopes and nesting.
3. Storage representations of data structures in definitional languages

A storage representation of data structures in definitional languages serves as a means of transfer of structures between program sub-processes, like iterations (loops). Often, transfer of the entire structure is unnecessary and undesirable (waste of storage and reduction in parallelism) and several techniques have been proposed for eliminating it. The best known, lenient cons [18] and I-structures [7] fill up the structure elements with special marks at the beginning of the program execution. Each special mark is replaced by the appropriate value as soon as the index and arguments for calculating this value become available. Any attempt to read (or ‘consume’) a structure element containing a special mark is delayed until the actual value is available. This means that if an operation attempts to use an element of a structure, it is delayed until that element has been computed.

Lists, streams and stacks are often implemented using pointers. Such a solution is convenient for language implementation [20] and eliminates copying of unchanged structure elements because a new structure is created either by linking a new element to the old structure or by setting a pointer to a part of the old structure. It should be stressed, however, that such a technique works only on structures with limited access (e.g. only the last element is accessible in lists or stacks) and cannot be applied to more general structures. Other disadvantages are that additional storage for pointers is needed and additional access time is spent on traversing a pointer’s chain.

In Hoffmann’s implementation of LUCID [23] an analysis of the use of variables in the scope of an iteration is performed, and if possible, only few elements (a window), instead of the entire vector, are assigned. The method was later extended to nested iterations [24]. However, no attempt was made to consider different scopes and order of nesting of iterations to achieve better memory utilization. The same problem of finding structures that can be represented by windows in main memory was considered in full generality in MODEL [31].

Both scope and nesting of iterations define the possibilities of reducing storage from a vector to a window. Therefore, in the MODEL language, neither the order of nesting nor the scope of iterations are defined by the user. One step in the compilation of a specification into the target program, called scheduling, is devoted to finding the optimal arrangement of the program sub-processes. Scheduling attempts to minimize the amount of storage for data transferred between loops. This amounts to decreasing memory allocation and execution time in one-processor machines [31]. It also corresponds to minimizing communication overhead in parallel architectures [29].

The optimization is achieved through a series of program transformations performed in the five following steps:

**Step 1. Syntax analysis** which produces an internal form of a specification and checks its syntactic correctness.

**Step 2. Source-to-source translation** of high level operations into elemental ones.
This translation produces lower level operations, which are later subjected to the optimization performed in the successive steps.

**Step 3.** *Array graph construction and analysis* which constructs the graph of data dependencies and performs analyses and checks of specification consistency and completeness.

**Step 4.** *Scheduling* that produces an ordered schedule of program sub-processes. It selects from many possible schedules the one that minimizes the memory allocation for object program structures.

**Step 5.** *Incremental structure optimization and code generation*, that produces the object program from the schedule.

Treating high level operations on structures as indivisible entities often leads to inefficient implementations. The result of the operation has to be entirely created before it could be used by other operations. Thus, parallelism is reduced and memory has to be assigned to the entire result structure [30]. Therefore, in Step 2, operations on high-level structures are decomposed (source-to-source) into operations on fields (individual elements). The source-to-source translation is performed as follows. First, a correspondence between the fields of the structures involved is established. The correspondence can be defined by the names of the fields or by the shapes of structures. Then a correspondence between subscripts of respective fields is found. Expressions relating indexes of arguments and result fields are derived from definitions of operations provided by the user (example of such definition is presented in Section 5). Finally, equations are generated for each field of the involved structures and declarations of interim variables (if any are necessary) are added.

The important advantage of this approach is that it leads to generality in the optimization method. It is not restricted to particular high-level data structure, such as stacks, but applies to any high-level structure whose translation into elementary operations produces incremental structures, including stacks, queues and arrays that change a constant number of elements in each operation. Also, the set of operations and functions available to the language users may be extended without affecting optimization algorithms. Newly added operations and functions will be optimized without additional effort (after this proper source-to-source translation). Moreover, optimization is possible independently of the representation chosen by the user (high-level or elemental operations).

In the MODEL system, a specification is represented by an *array graph*, where each node represents accessing, storing or evaluating of an entire structure and each edge represents dependencies among variables. The array graph is constructed in the third step of the MODEL compilation. The underlying graph of individual structure elements and their dependencies may be derived from the array graph based on the attributes of dimensionality, range, and forms of subscript expressions, which are given for each node and edge in the array graph. A node $A$ corresponding to an $m$-dimensional data or equation represents the elements from $A(1,1,\ldots,1)$ to $A(N_1,N_2,\ldots,N_m)$ where $N_1,\ldots,N_m$ are the ranges of dimensions 1 to $m$, respectively. Similarly, a directed edge represents all the instances of elemental
dependencies among the data elements of the nodes connected by that edge. The dependencies show precedence relations imposed on the execution order of the respective implied procedural program sub-processes. There are several types of such relations. A hierarchical precedence refers to the need to read in a structure before its components can be accessed, or vice-versa, the need to evaluate the components before a structure is stored away. Data dependency precedence refers to the need to evaluate the independent variables of an equation before the dependent variable can be evaluated. Similarly, data parameters of a structure (such as size of a dimension) must be evaluated before evaluating the respective structure.

Use of data dependency graphs to optimize programs, in particular for parallel execution, has been proposed recently in the literature (see for example [4, 16, 27, 41]). The distinctive features of the array graph of the MODEL language are a uniform representation of different types of dependencies, lack of control dependencies (flow of control is generated by the compiler) and presence of information about indexing expressions that are important in the analysis of iterative programs. In addition, representing entire arrays of data and dependencies by a single node or edge leads to manageable array graphs even for large (more than 10K lines) specifications. Owing to that, the optimization of generated programs, relying heavily on the analysis of indexing expressions, can be performed for real-life applications and not only for toy examples.

Scheduling is performed in the fourth step of the MODEL compilation. It starts by creating a component graph that consists of all the maximally strongly connected components (MSCC) in the array graph and the edges connecting the MSCCs. The component graph is therefore an acyclic directed graph. This graph can be topologically sorted, usually in many different ways that lead to different efficiencies of the generated object program. The objective of the scheduling is to find the topological sorting leading to the highest efficiency. This is done as follows.

First, the subscripts are determined for each node in the component graph. Iterations over these subscripts must bracket the respective nodes to define all the values of the elements in the array variables. Each node must be enclosed within nested iterations if the respective equations or data are of multiple dimensions. Next, attempts are made to enlarge the scope of these iterations. Nodes with the same range can be merged to form larger components. Merging scopes of iterations may enable sharing memory locations by elements of the same or related array variables. For instance let us consider a subgraph that consists of data nodes with subscripts over a common range, and of equations that use only subscript expressions of the form $I - k$, ($I$ denotes a subscript and $k$ a positive constant). If these nodes are all placed within the scope of a common range iteration loop, it is sufficient to retain in memory only a window of $k + 1$ elements in the respective dimensions. If it is possible to retain in memory only a window of the entire dimension of a variable, then the respective dimension is referred to as virtual, otherwise it is called physical.

Usually there are many ways in which components can be merged (for different dimensions), each corresponding to different topological sorting of the component
The memory requirements of different candidate scopes of iterations serves as the criterion for selecting the optimal merging. The selection is equivalent to NP-complete problem of finding a clique with the maximum weight of nodes in an undirected graph [10]. Therefore a heuristic is used as follows.

In each step the algorithm merges components with the same or related (sublinear or sawtooth [30]) dimensions. Since typically nodes and components have several dimensions, there would be many candidates for bases of the mergers. The compromise is to find the lowest memory cost of mergers progressively. First, dimensions and scopes of outside iterations are determined. Then, proceeding with one iteration at a time, each already processed iteration body is scheduled, and so forth. This procedure determines both the iteration scopes and the order of nesting. The procedure may not lead to the best solution, because the outer iteration scopes are determined without analyzing the possible influence of the inner iterations on memory cost. The inner iteration scopes are optimized locally within the outer ones. However, this greatly reduces the number of alternatives that need to be considered.

The selection of the seed component for further mergers may also effect the selected scopes and nesting of iterations. A seed component can be picked at random and then merging can proceed with components connected to the seed by edges. Considering all orders of merging would also greatly increase the number of alternatives. However, the order of merging has little effect on large iteration scopes—which are the most important. Therefore the merging along the edges departing from the seed was selected as a good compromise.

Virtual dimensions are found by the present MODEL compiler only if the subscript expressions used to reference the given variable are of the form \((I - k)\), where \(k\) is 0 or a positive integer, and \(I\) is a subscript or indirect index satisfying certain properties (the so-called sublinear or sawtooth indirect index, see [30]). It should be noted that elements of a window of size greater than 1 have to be shifted at the end of each loop execution, i.e. the first element shifted to the second position, the second to the third, etc. Alternately, the indexes referring to elements in the window may be cyclically shifted.

Use of windows is of utmost importance for optimization purposes. Particularly important is the case of window of size two, corresponding to indexing expressions of the form \(I - 1\). Such dimension is called a historical one, because it is used in each iteration step to distinguish the current value of a structure from the previous (historical) value. For a historical dimension, a part of the previous value is often transferred to a new one without any change. This is further discussed in the next section.

4. Optimization of stack-like data structures

As an example, consider a program for performing the following verbal specification: Take the value of \(A\), apply to it the function \(f\) ten times and place
the sum of the two last applications in $B$. A MODEL specification to that task is the following:

$$X(11) \text{ is field (NUM)};$$
$$X(1) = \text{if } I = 1 \text{ then } A \text{ else } f(X(I-1));$$
$$B = X(11) + X(10);$$

A naive implementation of this specification will allocate an array of 11 elements for $X$. The MODEL compiler, however, will realize that $X$ is used only as an intermediary between $A$ and $B$. Furthermore, it realizes that, once $X(I-1)$ is used for computing $X(I)$ and perhaps for the calculation of $B$, it is no longer needed, and hence a window of two elements, sliding over the virtual array $X$, is sufficient. Note that a window of one element is inadequate since $B$ needs two consecutive elements. Consequently, the MODEL compiler may produce the following program (or a very similar program) to execute this specification.

$$\text{DCL } XW(2) \text{ NUM;}$$
$$\text{DO } I = 1 \text{ to 11;}$$
$$XW(2) = XW(1);$$
$$XW(1) = \text{if } I = 1 \text{ then } A \text{ else } f(XW(2));$$
$$\text{if } I = 11 \text{ then } B = XW(1) + XW(2);$$
$$\text{END } \{ \text{DO } I \}. $$

This program consists of a declaration of $XW$ that represents a window of two elements. Inside the loop, the first statement copies into $XW(2)$ the value of $X(I-1)$ assumed to reside in $XW(1)$. The second statement computes in $XW(1)$ the value of $X(1)$, under the assumption that if $I > 1$ then $XW(2)$ contains the value of $X(I-1)$. The third statement places in $B$ the sum of the two recently computed elements, provided that $I = 11$ which signals the end of the loop. A special optimization may move this statement to the location following the loop.

Consider now the more complex case in which $A$ is an array of, say, 30 elements, and the function $f$ to be applied ten times to $A$ is an array function. Trying to represent the case of incremental data structures, the application of the function $f$ is assumed to modify only few of the 30 elements and leave the others unchanged. We may for example consider the specification:

$$A(30) \text{ is field (NUM);}$$
$$B \text{ is field (NUM);}$$
$$X(11, 30) \text{ is field (NUM);}$$
$$X(I, J) = \text{if } I = 1 \text{ then } A(J)$$
$$\text{else if } J = K \text{ then } X(I-1, L) + X(I-1, M)$$
$$\text{else } X(I-1, J);$$
$$B = X(11, 1) + X(10, 1);$$

In this specification $K$, $L$ and $M$ are integer expressions that may depend on $I$. $L$ and $M$ may also depend on $J$ but $K$ may not. For simplicity, assume that none of them depends on $J$. Observe that the intermediate array $X(11, 30)$ can be regarded
as a sequence of 11 one-dimensional arrays, $X(1,*), X(2,*), \ldots, X(11,*).$ Therefore, the same window identification technique applied before to the case of a simple $A$ can also be applied in the current case. Its straightforward application produces the following program:

```plaintext
DCL XW(2, 30) NUM;
DO I = 1 TO 11;
  DO J = 1 TO 30;
    XW(2, J) = XW(1, J);
  END {DO J}
DO J = 1 TO 30;
  IF I = 1 THEN XW(1, J) = A(J)
  ELSE IF J = M THEN XW(1, J) = XW(2, L) + XW(2, M)
  ELSE XW(1, J) = XW(2, J);
END {DO J}
IF Z = 11 THEN B = XW(1, 1) + XW(2, 1);
END (DO 1)
```

This program is obviously economical in memory usage, allocating an array of $2 \times 30$ instead of an array of $11 \times 30$ as would have been the case with the naive implementation. However, it is not very efficient in computation time. For each $I$, only one of the 30 elements of the array is recomputed, yet the program above copies twice the unchanged 29 elements for each $I$.

This is the point at which the special translation technique for incremental structures, suggested in this paper, is applied. It is triggered by equations of the form:

$$X(I, J) = \begin{cases} E_0(J) & \text{if } I = 1 \\ E_1(X(I-1,f_1(J)) & \text{if } J = S_1 \\ E_2(X(I-1,f_2(J)) & \text{if } J = S_2 \\ \vdots & \text{if } J = S_n \\ E_n(X(I-1,f_n(J)) & \text{if } J = S_n \\ X(I-1,J) & \text{else} \end{cases}$$

It is assumed that $S_1, \ldots, S_n$ do not depend on $J$ (but may depend on $I$). The expressions $E_k, k > 0$ may contain several occurrences of the form $X(I - 1, f_k(J))$ for different $f_k(J)$'s. Furthermore, we assume that all the $S_i, i = 1, \ldots, n$ are distinct and always within the subscript range allowed for the second dimension of $X$.

With these assumptions, the following special translation is suggested for such equations:

```plaintext
DCL XW(2, *)
DO I = 1 TO ...
  IF I = 1 THEN
    {Initial case}
    DO J = 1 TO ...
      XW(1, J) = E_0(J);
    END {I = 1}
```
As before, it is possible to move the case \( I = 1 \) out of the loop, which then starts at \( I = 2 \). Inside the loop for the general case, there is a saving section that saves in \( XW(2, *) \) all the old values of \( X(I - 1, *) \) which are needed for the computation of \( X(I, *) \). The saving section is followed by a computation section, which computed in \( XW(1, *) \) the values of the elements of \( X(I, *) \) that may have changed in comparison with \( X(I - 1, *) \).

We may illustrate this translation on the particular example studied above. It yields the following program:

```algebra
DO \( J = 1 \) TO 30;
    \( XW(1, J) = A(J) \);
END \{DO J\}
DO \( I = 2 \) TO 11;
{Save needed old values}
    \( XW(2, L) = XW(1, L) \);
    \( XW(2, M) = XW(1, M) \);
    IF \( I = 11 \) THEN \( XW(2, 1) = XW(1, 1) \);
{Compute new values}
    \( XW(1, K) = XW(2, L) + XW(2, M) \);
{Compute result if necessary}
    IF \( I = 11 \) THEN \( B = XW(1, 1) + XW(2, 1) \);
END \{DO \( I \)\}
```

Note that since \( K, L \) and \( M \) may be expressions dependent on \( I \), the above program is close to the best program possible for this specification.

The range of applicability of this translation can be extended in several ways. First we may consider equations of the more general following form:

\[
X(I, J) = \text{IF } I = 1 \text{ THEN } E_0(J)
\]

\[
\text{ELSE IF } J = S_1 \text{ AND } C_1(X(I - 1, g_1(J))) \text{ THEN } E_1(X(I - 1, f_1(J)))
\]

\[
\text{ELSE IF } J = S_n \text{ AND } C_n(X(I - 1, g_n(J))) \text{ THEN } E_n(X(I - 1, f_n(J)))
\]

\[
\text{ELSE } X(I - 1, J);
\]

Here, we have to save in \( XW(2, *) \) also the elements needed for the evaluation of the conditions \( C_1, \ldots, C_n \). Consequently, for each \( i = 1, \ldots, n \) in the equation,
the saving section in the translation should contain the statements:

\[ XW(2, g_i(S_i)) = XW(1, g_i(S_i)) \]

If \( C_i(XW(1, g_i(S_i))) \) then \( XW(2, f_i(S_i)) = XW(1, f_i(S_i)) \)

Similarly, for each such \( i \), the computation section should contain the statement:

\[ \text{IF } C_i(XW(2, g_i(S_i))) \text{ THEN } XW(1, S_i) = E_i(XW(1, f_i(S_i))) \]

Note that we cannot save \( XW(1, f_i(S_i)) \) without first checking that \( C_i(XW(1, g_i(S_i))) \) holds. This is because if \( C_i \) does not hold, \( f_i(S_i) \) may be out of the subscript range of the second dimension or even undefined.

Another extension is that we may have several definitions of this form. We may have, for example, one such equation defining \( X(I, *) \) in terms of \( X(I - 1, *) \) and \( Y(I - 1, *) \) needed for the computation of both \( X(I, *) \) and \( Y(I, *) \).

A possible additional optimization that has not been implemented yet, is to define \( XW \) as a singly dimensioned array. Since the number of temporary variables that are now saved in \( XW(2, *) \) is small, we may allocate instead several special variables and use them to store the old values that need to be saved. This leads to a further decrease in the memory requirements of the translated program.

The optimization algorithm consists of the following three steps:

**Step 1.** Analysis of conditional equations inside loops.

**Step 2.** Use-definition chain analysis [3] for the optimized variables.

**Step 3.** Modification of the generated object program.

The analyses (Step 1 and 2) are applied to the schedule (cf. Schedule Report discussed in section 5.2) produced by the MODEL compiler. Both steps are repeated recursively for nested loops. The results of these analyses are used during the generation of the object program to modify the code for the affected structures and loops. Since the optimization is performed on the schedule that represents a typical iterative program, the method can be easily applied also to programs that are generated by other definitional language compilers, such as Hiobol [38], Lucid [8], MEDEE [17], and Xloop [41].

5. Illustration of the method by an example

5.1. Example specification

As mentioned in the previous section, the MODEL system performs global analysis of the specification to determine optimal mapping of the source structures into the object program arrays. This process is illustrated below through an example, which describes iterative implementation of Ackermann's function [32] defined as:

\[
f(m, n) = \begin{cases} 
  n + 1 & \text{if } m = 0, \\
  f(m - 1, 1) & \text{if } m \neq 0 \& n = 0, \\
  f(m - 1, f(m, n - 1)) & \text{if } m \neq 0 \& n \neq 0.
\end{cases}
\]

An iterative algorithm for evaluation of this function is given by Manna [32] (cf. program in Fig. 1). It uses a stack to represent a sequence of recursive activations...
start; \( S(1) \leftarrow m; S(2) \leftarrow n; i \leftarrow 2; \)

loop: if \( i = 1 \) then begin \( f \leftarrow S(1); \) halt end
else if \( S(i-1) = 0 \) then begin \( S(i-1) \leftarrow S(i-1) + 1; S(i) \leftarrow 1; \) goto loop end
else if \( S(i) = 0 \) then
  begin \( S(i-1) \leftarrow S(i-1) - 1; S(i) \leftarrow 1; \) goto loop
  end
else begin
  \( S(i+1) \leftarrow S(i) - 1; S(i) \leftarrow S(i-1); S(i-1) \leftarrow S(i-1) - 1; i \leftarrow i + 1; \) goto loop end

Fig. 1. Iterative algorithm for Ackermann's function.

Each time the second argument of the Ackermann's function is defined by the recursive call to \( f \), a new pair of arguments is pushed onto the stack. A pair of arguments is popped off the stack when a corresponding value of the function \( f \) is defined directly (i.e. the first argument is zero).

A MODEL specification for this computation uses a sequence of stacks \( (S) \). Each stack is composed of a vector of elements \( (E) \) and a scalar \( (T) \) pointing to the top of the stack, as defined in Fig. 2. The input file INP consists of one record \( REC \) that contains two numeric fields storing values of parameters \( M \) and \( N \), respectively. Similarly, the output file OUT consists of one record that contains a numeric field with the result value \( F \). \( S \) is a one-dimensional structure that consists of a scalar \( T \) and a vector \( E \). \( T \) denotes the size of the vector \( E \) and may also serve as a pointer to the highest element of \( E \). \( E \) contains the elements of the stack. \( E \) is a two-dimensional array and its first dimension is inherited from its ancestor \( S \). Both dimensions of \( E \) are of variable size (as indicated by the asterisk * following the declarations of \( S \) and \( E \)). For each instance of \( S \) there is one instance of \( T \) and a vector of values of \( E \). Both sizes of dimensions of \( E \) depend on the values of the function parameters \( m \) and \( n \) in a complex way. The size of the vector \( E \) (i.e., the stack size) is different for different indexes of \( S \). Each stack instance \( S(I) \) represents one evaluation of \( f \) applied to the highest two elements of the preceding stack. The subscript \( I \) indexes a historical dimension that would not be used in a procedural language. This is further explained when the algorithm for computing \( f(m, n) \) is discussed.

The MODEL language incorporates many functions that operate on compound data structures. The operations \( \text{PUSH} \text{element, stack_expression} \) and \( \text{POP} \text{stack_expression} \) return a stack. A constant \( \text{EMPTY_STACK} \) denotes the empty stack. The function \( \text{TOP} \text{stack_expression} \) returns the top element of the stack. The algorithm for computing \( f(m, n) \) can thus be written as in Fig. 3. The

```
1 INP IS FILE, /* input file with parameters */
2 REC IS RECORD,
3 M IS FIELD (NUMERIC(4)), /* the first parameter */
3 N IS FIELD (NUMERIC(4)); /* the second parameter */
1 OUT IS FILE, /* output file with function value */
2 REC IS RECORD,
3 F IS FIELD (NUMERIC(4)); /* function value */
1 S(*) IS GROUP, /* stack vector */
2 T IS FIELD (NUMERIC(8)), /* index of the top of stack */
2 E(*) IS FIELD (NUMERIC(8)); /* stack elements */
```

Fig. 2. The data declarations in the specification of the example.
first statement defines the new (current) stack in terms of the previous one. Initially the stack contains parameters of the function being evaluated (in fact they are pushed on the empty stack). For the next steps a new stack is defined following the above given description.

The range of a dimension can be referred to in MODEL by prefixing the name of the respective structure with the keywords END or SIZE. The END variable has the same shape as the one named in its suffix. All its elements have value 0, except the last one, which is equal to 1. The last statement in Fig. 3 defines how to recognize the last vector of S (it will be used in the generated object program as the condition for terminating the iteration associated with the subscript I). The second statement in Fig. 3 uses SIZE to determine the iteration step in which the parameters initially pushed on the stack have been replaced by the function value (it is the only element of the stack in that instant, so it is also the top of the stack).

5.2. Decomposition of high-level structure operations into elementary operations

Decomposition of high-level operations and functions, like POP, PUSH or TOP, is performed through their source-to-source translation into operations on fields (structure elements). For the specification in Fig. 3, the source-to-source translation is based on the following definitions of stack operations:

1. Let a stack S_STACK be, like in our example, represented by a structure S that consist of a scalar \( T \) defining the size of the stack and a vector of stack elements \( E(J) \), where \( J = 1, 2, \ldots \). \( T \) is a subscript.

Let \( S_1 \) and \( S_2 \) be representations of stacks \( S_1_STACK \) and \( S_2_STACK \), respectively. Let \( T_1, E_1 \) and \( T_2, E_2 \) denote the respective substructures of \( S_1 \) and \( S_2 \).

2. If \( S_1_STACK = PUSH(x, S_2_STACK) \) then \( T_1 = T_2 + 1 \), and \( E_1(J) = x \), if \( J = T_1 \) and \( E_1(J) = E_2(J) \) otherwise.

3. If \( S_1_STACK = POP(S_2_STACK) \) then \( T_1 = T_2 - 1 \) and \( E_1(J) = E_2(J) \), for all \( J \leq T_1 \)

4. \( TOP(S_1_STACK) \) is equal to \( E_1(T_1) \).

5. If \( S_1_STACK = EMPTY_STACK \) then \( T_1 = 0 \).
Based on these definitions, the MODEL system automatically derives elementary equations and termination conditions for the translated specification, as presented in Fig. 4. Interim variables created for function superposition can be eliminated in the process of the translation [30]. It should be noted that the similar elementary specification would be obtained, if one would write it directly from the iterative algorithm of Manna presented in Fig. 1.

For the first equation in Fig. 3, decomposition produces three equations: two for the fields of structure \( S \) (\( T \) and \( E \)) and one for the range of dimension \( J \) (defined through variable \( SIZE.E \)). Equation (1) defines the top \( T \) of the stack. Each \textsc{pop} causes a decrease of \( T \) by one and each \textsc{push} increases \( T \) by one. Equation (2) states the definition of \( S \) in terms of the individual stack elements. Equation (3) defines the size of the stack vector. It uses the \textit{SIZE} variable that defines the range of the dimensions indexed by \( I \). The name of such a variable consists of the prefix \textit{SIZE} and the name of the respective structure. The \textit{SIZE} variable has one dimension less than the structure named in its suffix. Equation (3) states that the second dimension of \( E(I, J) \) has \( T(I) \) elements.

5.3. Array graph

The array graph of the decomposed specification is presented in Fig. 5. Three different types of edges have been shown: data dependency implied by equations.

```/* declare representation of the stack \( S \) */
J IS SUBSCRIPT; /* \( J \)—subscript of the stack vector */

/* define index of the top of the stack */
(1) \( T(I) = \) IF \( I = 1 \) THEN 2 ELSE IF \( E(I-1, T(I-1)-1) = \) 0 THEN \( T(I-1)-1 \)
ELSE IF \( E(I-1, T(I-1)) = \) 0 THEN \( T(I-1) \)
ELSE \( T(I-1)+1; \)

/* define stack contents */
(2) \( E(I, J) = \) IF \( I = 1 \) THEN IF \( J = \) \( N \) ELSE \( M \)
ELSE IF \( E(I-1, T(I-1)-1) = \) 0 THEN IF \( J = T(I) \) THEN \( E(I-1, J+1)+1 \)
ELSE \( E(I-1, J) \)
ELSE IF \( E(I-1, T(I-1)) = \) 0 THEN IF \( J = T(I) \) THEN 1
ELSE IF \( J = T(I)-1 \) THEN \( E(I-1, J)-1 \)
ELSE \( E(I-1, J) \)
ELSE IF \( J = T(I) \) THEN \( E(I-1, J-1)-1 \)
ELSE IF \( J = T(I)-1 \) THEN \( E(I-1, J)-1 \)
ELSE IF \( J = T(I)-2 \) THEN \( E(I-1, J)-1 \)
ELSE \( E(I-1, J); \)

/* define range of the stack vector */
(3) \( SIZE.E(I) = T(I); \)

/* define result */
(4) IF \( SIZE.E(I) = 1 \) THEN \( F = E(I, T(I)); \)

/* define range of stack’s sequence */
(5) \( END.S(I) = (SIZE.E(I) = 1); \)
```

Fig. 4. Decomposed elementary equations for specification in Fig. 3.
data parameters implied by the semantics of \textit{END} and \textit{SIZE} variables, and data hierarchy implied by the source and target data declarations. In addition, edges attributed with subscript expressions different from simple subscript $I$, are marked with those expressions. The equation nodes are denoted by $e$ followed by the equation number.

5.4. Scheduling

The schedule of sub-processes of the object program (i.e. a skeletal form of the object code) produced for the presented specification by the MODEL compiler is presented in Table 1. There are two iterations in the schedule. The outer main loop corresponds to the subscript $Z$ and contains almost the entire program. The inner loop corresponds to the subscript $Z$ and performs operations on stack elements.

The MODEL compiler has reduced the sizes of dimensions indexed by the subscript $I$. The structure \textit{SIZE.S} is declared in the specification as a vector. However, since it is always used with the subscript $I$, it got assigned a virtual dimension in the object program and was declared there as a scalar. Both, $T$ and $S$, are indexed by $I - 1$ and therefore they require a window of size two for the subscript $I$. All the references to these variables in the generated object program will be modified. For
Example, the conditional assignment in equation (2):

\[ E(I, J) = E(I - 1, J) \]

in the generated object code is translated into

\[ E(2, J) = E(1, J). \]

Note that references in equation (2) are made to future and previous values of the subscript \( J \) (e.g. \( J - 1, J + 1 \)). Consequently, the second dimension of the structure \( E \) is found to be physical. The object code structure declarations in PL/1 are listed in the Appendix.

5.5. Further optimizing operations to avoid unnecessary copying

The inner loop of the program generated for specification of the Ackermann’s function contains only a single equation defining the vector \( E \). As this equation conforms with the general form presented in Section 4, the method described in that section can be applied and would lead to complete removal of the inner loop. However, for the structure \( E \), it is still necessary to retain the double vectors \( E(1, J) \) and \( E(2, J) \) in memory for the following reason. A part of the equation (2) defining
E(I, J) is:

E(I, J) = \ldots \text{if } J = T(I) - 1 \text{ then } E(I - 1, J - 1)

\text{else if } J = T(I) - 2 \text{ then } E(I - 1, J) - 1 \ldots

Both elements, E(I, T(I) - 1) and E(I, T(I) - 2), are defined by the same element E(I - 1, T(I) - 2). The latter cannot be overwritten by E(I, T(I) - 2). Therefore the object code produced for this case is:

E(2, T(I) - 2) = E(1, T(I) - 2); \quad \text{copying to the temporary storage}

E(1, T(I) - 2) = E(2, T(I) - 2) - 1; \quad \text{assignment of}

E(1, T(I) - 1) = E(2, T(I) - 2); \quad \text{the new values}

Note that the object vector S(2) is used merely as a temporary storage and it contains only few elements, not the entire vector. A further optimization would reduce it to a smaller array containing only three elements. Note also that the need for the temporary storage and elements that have to be copied there, are automatically detected through analysis of equation (2) defining E (comp. Section 4).

Since the size of the stack in the given example grows very rapidly with the increase of the parameter values, the benefits of the loop removal are particularly spectacular. The transformed PL/I program for the specification of the Ackermann’s function is presented in the Appendix.

6. Conclusions

The major disadvantage that impedes general acceptance of definitional languages is the inefficiency caused by straightforward interpretation of the user’s specification. Attempts to provide users of definitional languages with special direct control to regain efficiency spoils the simplicity of these languages and decreases ease of their use. For example, in a technique for efficient management in functional languages [21] the user is responsible for determining which arrays should be stored. Data-flow techniques like lenient cons and I-structures [1] are applicable only when a data structure is stored and accessed sequentially. Thus, they are not adequate for structures which are stored and accessed alternately. Those includes stacks, which are important for automatic recursion removal [5].

The approach presented in this paper obtains efficiency of execution by global deep analysis of the specification and automatic detection of restricted usage of structures. The method of optimization is based on motion of code out of loops and empty loops removal. A novel technique of detection of movable statements is applied. Owing to it, incremental structures can also be optimized.

Global analysis of the user’s specification performed by the MODEL compiler (results of which are available during compilation) helped in implementing checks of correctness and applicability of the described program transformations. It should
be stressed however that the techniques of this global analysis are general and may be used by any definitional language translator.

The implementation of the presented algorithm led to a significant increase in efficiency of the object programs that use incremental data structures. Those included stacks (Ackermann’s function and variety of recursive algorithms), queues (resource allocation program [37]) and multidimensional arrays (topological sort, backtracking algorithms [40], and various business application programs). As an example, we compared the performance of the programs generated with and without the discussed optimization for the above specification of the Ackermann’s function. Even for small arguments, (3,3) to (3,5), the transformed object program was several (8 to 32) times faster than the original one. The gain of efficiency grows rapidly with the value of arguments. The optimization stage consumes a small percentage of the total compilation time. Thus, the compilation time overhead is negligible. The presented program compiled on VAX-11/750 under VAX-VMS in 37 sec, out of which 1.5 sec (i.e. about 4%) was spent on the extra optimization. This can be contrasted with the reduction in execution time reported below.

<table>
<thead>
<tr>
<th>Arguments ((m, n))</th>
<th>3, 3</th>
<th>3, 4</th>
<th>3, 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original program</td>
<td>194.88</td>
<td>1481.20</td>
<td>14322.07</td>
</tr>
<tr>
<td>Optimized program</td>
<td>25.53</td>
<td>102.84</td>
<td>444.32</td>
</tr>
<tr>
<td>Reduction in seconds</td>
<td>169.35</td>
<td>1378.36</td>
<td>13877.75</td>
</tr>
<tr>
<td>Factor (orig/opt)</td>
<td>8</td>
<td>15</td>
<td>32</td>
</tr>
</tbody>
</table>

Appendix

To illustrate the transformations made during the optimization the generated object program for the given above specification is presented. As previously described, the program is generated according to the schedule in Schedule Report. The details, unimportant for the optimization process, have been replaced by comments in the listing of the object program presented below. The biggest transformation is done to equation (2). To establish the correspondence between the initial code and the transformed one, below we have labeled parts of this equation.

\[
\begin{align*}
/*\text{IF-1*/} & \quad E(I, J) = \text{IF } I = 1 \\
& \quad \text{THEN IF } J = 2 \text{ THEN } N \text{ ELSE } M \\
/*\text{IF-3*/} & \quad \text{ELSE IF } E(I - 1, T(I - 1) - 1) = 0 \\
& \quad \text{THEN IF } J = T(I) \\
/*\text{THEN-4*/} & \quad \text{THEN } E(I - 1, J + 1) + 1 \\
& \quad \text{ELSE } E(I - 1, J) \\
/*\text{IF-5*/} & \quad \text{ELSE IF } E(I - 1, T(I - 1)) = 0 \\
/*\text{THEN-5*/} & \quad \text{THEN IF } J = T(I) \text{ THEN } 1 \\
/*\text{THEN-7*/} & \quad \text{ELSE IF } J = T(I) - 1 \text{ THEN } E(I - 1, J - 1) \\
& \quad \text{ELSE } E(I - 1, J)
\end{align*}
\]
Because old stack elements are needed for the computation of new elements at the same position, \( E \) is represented by a double vector (window of size \( k + 1 \) for \( k = 1 \)). Moving the old element to the temporary storage, (the vector \( s.e(2, s.I) \)) is the first part of the transformed equation (2). The second part, using the same conditions as the first one, executes assignments implied by the equation (2). The inner loop of the program has been completely removed.

The optimized object program is presented below.

```
ACM: procedure OPTIONS(MAIN);
/... declarations of input and output file, omitted here*/
DCL $NOT_DONE(20) BIT(1);
DCL 1 INP,
   2 REC,
      3 M PIC'99999',
      3 N PIC'99999';
DCL 1 OUT,
   2 REC,
      3 F PIC'99999';
DCL 1 S,
   2 T(2) PIC'99999999', /*window of size 2*/
   2 E(2,9999) PIC'999999999'; /*second dimension is physical*/
DCL 1 INTERIM,
   2 ENDS$S(2) BIT(1),
   2 SIZES$E FIXED BIN; /*virtual dimension (of size 1)*/
DCL ($I1,$I2) FIXED BIN; /*corresponding to subscripts \( I, J \) respectively*/
/... opening and reading the input file into \( N \) and \( M \), omitted here*/
$I1 =0;
$NOT_DONE(1) = '1'B;

/* main loop for subscript \( I \)*/
do while($NOT_DONE(1));
   $I1=$I1+1;
   if $I1-1 then S.T(2) = 2; /* equation (1) */
   else if S.E(1, S.T(1)-1) = 0 then S.T(2) = S.T(1)-1;
   else if S.E(1, S.T(1)) = 0 then S.T(2) = S.T(1);
```
\begin{verbatim}
ELSE S.T(2) = S.T(1) + 1;
SIZE$S_E = S.T(2); /* equation (2) */
/* transformed inner loop for subscript J */
/* moving needed element into the temporary
storage in preparation for equation 2 */
IF ^($I1 = 1) THEN DO;
S.E.(2, S.T(1) - 1) = S.E(1, S.T(1) - 1); /* if-3, then-4 */
IF S.E(2, S.T(1) - 1) = 0 THEN S.E(2, S.T(2) + 1) = S.E(1, S.T(2) + 1); ELSE DO;
S.E(2, S.T(1)) = S.E(1, S.T(1)); /* if-5 */
END;
IF S.E(2, S.T(1)) = 0 THEN S.E(2, S.T(2) - 1) = S.E(1, S.T(2) - 1); /* then-7 */
ELSE DO;
S.E(2, S.T(2) - 1) = S.E(1, S.T(2) - 1); /* then-8 */
S.E(2, S.T(2) - 2) = S.E(1, S.T(2) - 2);
END;
END;
/* transformed equation (2) */
IF ($I1 = 1) THEN DO;
S.E(1, 2) = INP.N; /* if-1 */
DO $I2 = 1 TO SIZE$S_E;
IF ^($I2 = 2) THEN S.E(1, $I2) = INP.M;
END;
END;
IF ^($I1 = 1) THEN IF S.E(2, S.T(1) - 1) = 0
THEN S.E(1, S.T(2)) = S.E(2, S.T(2) + 1) + 1; /* then-4 */
ELSE IF S.E(2, S.T(1)) = 0 THEN /* then-5 */
DO; S.E(1, S.T(2)) = 1;
S.E(1, S.T(2) - 1) = S.E(2, S.T(2) - 1) - 1;
END;
ELSE DO; /* else-5 */
S.E(1, S.T(2)) = S.E(2, S.T(2) - 1) - 1;
S.E(1, S.T(2) - 1) = S.E(2, S.T(2) - 2);
S.E(1, S.T(2) - 2) = S.E(2, S.T(2) - 2) - 1;
END;
IF SIZE$S_E = 1 THEN OUT.F = S.E(1, 1); /* equation (4) */
END$S = (SIZE$S_E = 1); /* equation (5) */
IF END$S(2) THEN $NOT_DONE(1) = 'O'B; /* termination condition of main loop */
\end{verbatim}
END$S(1) = END$S(2); /* shifting window variables */
S.T(1) = S.T(2);
END;
/* . . . writing and closing the output file, omitted here */
RETURN;
END ACM;

References

Efficient handling of data structures in definitional languages

[38] G.R. Ruth, Data driven loops, MIT/LCS/TR-244, 1981.