Frederic N. Ris

Experience with Access Functions in an Experimental Compiler

This paper describes an access function subsystem embedded in portions of an experimental microcode compiler which was built and used during 1973–6 using the IBM PL/I optimizing compiler under VM/370 and CMS. The use of the access function subsystem in this context was itself an experiment, performed by a group for all of whom PL/I was a new language and VM/370 a new operating system. The implementation of the subsystem was done strictly within the confines of the PL/I language. The basic objectives were ease of use, provision of a focal point for global storage management, extensive run-time validity checking with appropriate diagnostics, and data protection. Beyond satisfying these objectives, the subsystem proved more valuable than anticipated due to positive contributions made to debugging code in the VM/370 interactive development environment.

Introduction

An *access function* is a stylized procedure that provides random-access data to an application program which requires them for purposes other than storage management. Transparently to the calling application program, access function programs manage the storage of data and determine validity and authorization of data requests. In general, two procedures are applicable to a particular datum: one which stores it and one which retrieves it.

This paper describes experiences in the design and use of access functions in a PL/I programming environment by a group implementing an experimental compiler whose aim was to produce highly optimized code for a variety of vertically microprogrammed machines. The prototype microcode compiler contained many interconnected data structures naturally representable in tabular form. Particular table columns might contain arithmetic or string values, references to rows in the same or another table, and lists or sets of these elements.

We consider here a data base consisting of a number of two-dimensional tables having logical interconnections within tables as well as among them. Each table has associated with it a fixed number of statically named columns with predefined data attributes, and a dynamically varying number of rows, each with a uniquely generated name, an instance of which is an *entry* in its particular table. A particular access functional reference consists of the function name (explicitly identifying which column) and the table entry parameter (explicitly identifying the row for which the value in the named column is desired, and thus implicitly identifying the table as well).

In terms of the programming language in which the application is written, the access functions are external procedures. Each table entry (row identifier) is represented by some data type in the language (POINTER in the present case), and the table columns and functional values will be drawn from the basic data types in the language (FIXED BINARY (31), CHARACTER (32), . . .), including the one used to represent table entries.

By contrast, an "access method" is a collection of programs which manage the transfer of strings of bits from one storage medium to another, often dealing with a physical organization on one side of the interface and a logical organization on the other. With an access method it is still up to the application program to interpret fields within the logical record. In most practical situations storage hierarchies cannot be ignored, and they must deal with strings of

bits representing programs as well as data. The subsystem described here was operated in a virtual store and dealt only with relatively simple data forms.

If a large proportion of data is representable conveniently in a tabular form such as that described above, it is possible to make all accesses solely through a regular procedural interface. The code can then be separated into two pieces: that part above the interface, which performs the intended services for the application, and that part beneath, which manages the storage of data and implements the services provided at the interface. This latter ensemble of code we refer to as the *access function subsystem.*

In developing the prototype microcode compiler we used the access function subsystem interface to separate organizational questions of storage management and data structure representation from the compiler algorithms of interest. Functional code above this interface could create, destroy, order, and access all global data through some two dozen generic procedures and several hundred item-specific operations. The structure of the code which implemented the access functions beneath the interface was extremely regular so that its generation could easily be mechanized. Binding of compiler algorithms to data structures was thereby done at program load time, avoiding both (1) recompilation of functional code as tables expanded, and (2) heavy execution-time penalties which would have arisen from a dynamic associative storage organization. Thus the structured code in the functional half of the ensemble was complemented by a structured data management supported by the access functions. All code on both sides of the interface was written in PL/I.

When we set out to do this experiment in early 1973, PL/I was the implementation language of choice due to its broad functional flexibility. We had no desire to do language development and were curious to see how far we could get PL/I to carry us, given that the language was not designed with this sort of use explicitly in mind. Much programming language research activity in the 1970s centered around design issues for the support of "data abstraction," "structure hiding," "data type encapsulation," and "packages with controlled export" in languages such as CLU [1], Euclid [2], and Modula [3], and culminating in the Ada language [4], for which one of the procurement requirements was precisely this capability [5–9].

Basic objectives of access functions

- *Structured data management*

If the logical data organization defined by such an interface is appropriate to the application, this division can make a substantial contribution to reducing the design complexity of the system as a whole. In parallel with a well-structured program design above the interface, a well-structured data management subsystem can be used beneath to implement it, and the execution overhead induced by the presence of the interface itself will exact a reasonable efficiency penalty in return for a faster, cleaner development and more easily maintained code. The division allows the data structures viewed from above to be drawn as tables in which entries can be created, destroyed, and moved around with ease. The implementation decisions, many of which are irrelevant to the application, can then be isolated and concentrated in the access function subsystem.

In particular, storage management for the tables can be completely subsumed in the subsystem. Allocation strategy can range from fixed partitions established when the functions are compiled to a completely dynamic allocation transparent above the interface and limited only by the total size of storage available at execution time. Spilling of excess data to lower levels of a storage hierarchy is also possible, although we did not require this and therefore did not attempt it.

- *Ease of use*

For this strategy to be effective, the functions themselves must be easy to use. The source language from which they are deployed must allow such data access via a natural, functional notation. Name qualification requirements must be flexible enough to permit minimal qualifications consistent with reliability. The entry cost to the programmer must be low. There will be a mass of detail in the interface, but he need know only a small fraction of it at any one time. That fraction which is required at the outset must be necessary for his needs, and it should also be sufficient. Since the table structures are being conceptually manipulated analogously to the way in which pictures drawn during design are manually manipulated, these operations should be naturally reflected in the basic functional structure of the interface. Finally, since questions of storage management are handled in the subsystem, the programmer above the interface need not deal in imponderable parameters of storage strategies which require raw information not otherwise available or useful to his application.

- *Error prevention and checking*

Another motivation for an access function interface is that erroneous patterns of data usage above it can be checked and prevented by the code beneath much more easily. Function requests can be validated against properties of the entry (is the function appropriate to the particular table?) or against other fields present in the entry (does it ask for the fifth element in a three-element list?). A uniform error-handling protocol within the access function subsystem can provide maximum information and control flexibility because the

overhead is widely distributed over the function of the entire subsystem.

Data protection can be extended even further by a passive authorization check of whether a particular process is allowed to access or update particular fields or entries. More importantly, unused tables and functions are completely transparent to the application program, so the wrong field in a structure cannot be accessed by mistake unless explicitly named. Further, if tables grow by adding columns, recompilation will not be necessary for those programs which do not require the information they contain. Version synchronization is thus less of a problem.

Motivations and environment

The access function subsystem described in the next section played a role in four phases of an experimental compiler. Its object was the generation, from a subset of PL/I (representative of systems programming), of microcode for the IBM 3145 Processor, on which the System/370 Model 145 is implemented. Source language translation, dictionary construction, semantic interpretation, consistency checks, and diagnostic message generation were performed by the PL/I Checkout Compiler. The text produced (normally destined for interpretation) was transformed into our starting text level by conventional code generation techniques, assuming an infinite number of registers. The areas of principal interest in our experimental model were then (1) preliminary optimizations of incoming text, (2) transformation of this abstract text into equivalent sequences which reflect the data flow paths available on the target machine, (3) allocation and assignment of machine registers to the resulting text, and (4) generation of microcode, taking into account the register assignments, maintaining machine state consistency with respect to register addressability, and the like.

We wished to develop each of these phases independently of the others to the greatest extent possible and so we defined external representations of the text levels connecting them. Each phase became a single program which read its input in the external form, did its work, and output the resulting text for the next phase. This method of operation allowed easy generation of test data for each phase and provided a means of simulating incompletely implemented functions by hand, through the use of a text editor on the intermediate external forms.

The same access function subsystem was made equally available to all phases, and those tables common to more than one phase were accessed in the same way in each phase. While this helped document the communications between phases, extensive programming was required to produce the input and output routines for each phase which transform the external texts into the internal tables and *vice versa*.

This division of compiler function into completely separate programs was a conscious trade-off of ultimate program efficiency for design and implementation independence at the highest level of compiler structure. Since the prime objective was to establish feasibility of proposed techniques for generating microcode rather than to achieve fast compile time, broad experiments were attempted with the implementation, of which the access function subsystem was the most pervasive.

We provided for a large amount of global data. In addition to the tables containing text and its dictionary of operands and labels in each phase, other tables would describe the properties of text operators in object machine terms to enable one to select potential code generation patterns. The entire register space of the object machine was represented in one of the tables. Finally, the register allocation and assignment process required a great deal of control and data flow information, and this was provided using the same tabular formulation.

We felt that our outlook on the logical relationships among target machine characteristics, software convention requirements, and actual program text would coordinate well with our proposed algorithms for microcode generation. Nonetheless, we desired a straightforward representation of this large amount of data, not all the details of which were known *ab initio*, to facilitate coherent expression of our algorithms, assisting both the initial design and subsequent experimental modifications we knew we would want to make. Because we were concentrating on the object code generation aspects of compilation rather than the better-understood translation and optimization components, we wished to focus on the algorithms which provided the function rather than representation which provided efficiency.

Performance was critical insofar as our own programming and testing time was concerned, but never so critical as to necessitate major design efforts whose sole purpose was to enhance performance at the expense of complicating the way in which the function of the compiler was carried out. Because the work was being done in a virtual store, it was decided early that no conventional program text spill mechanism would be installed. Rather, text would be accessed directly and the virtual machine size would be used to control the amount of storage available. Thus, small programs could be compiled in small virtual machines while large programs would require larger virtual machines.

- *Expected benefits*

We anticipated benefits in the design, coding, and testing of the compiler code. At the design level the structural complexity of the algorithms would be commensurate with the functions performed by and the requirements of individual programs.

At the time of coding, we expected to enlist the PL/I compiler's help in providing syntactic safeguards against inadvertent errors and to maximize the probability of getting a diagnostic message when a misuse occurred.

During testing, we hoped to reduce debugging time with early and comprehensive diagnostics when the data were misused. The transparency of unneeded table information would keep recompilation costs to a minimum and would prevent many errors from occurring in the first place.

Insofar as representation of particular table items was concerned, we attempted to keep explicit encodings as much in the background as possible. This was expected to facilitate program debugging and algorithm experimentation by making output of forms intermediate to a phase easier to produce, as well as giving the access function diagnostic messages a better chance of being meaningful. Here we were trading storage for function to keep the implementation process running smoothly.

- *Practical requirements*

Several practical requirements arose from these considerations. First, because we would be dealing with program text which must be expanded, contracted, and moved around, one had to be able to insert, delete, and rearrange the order of table entries (rows in the tables) with language which was direct and which could be implemented with complete reliability.

Many of our algorithms were couched in terms of operations on sets: for example, requiring the enumeration of all variables used in a statement, the set of statements from which a branch to a labeled statement might be made, or the set of machine registers conformable with the storage requirements of a particular variable at a particular statement. An early consideration was therefore that we should provide a means of dealing with sets from an arbitrary universe.

As the rest of the compiler was to be written in PL/I, we wished to implement the access function subsystem in strictly legal PL/I, because it was recognized that a great deal of code would have to function correctly and with little maintenance for long periods of time, during which there might be changes in the compiler. We thus had consciously to avoid surreptitious use of "variant records," as described below.

Augmentation of the subsystem had to be efficient. It had to be possible to add new entries to existing tables quickly and to generate entirely new table formats almost as easily. We did not want to have to recompile any but those programs directly concerned when such routine table modifications were made.

Finally, it was clear that access was rather more important than update, so the chief emphasis was placed on information retrieval rather than storage.

Implementation

- *The basic structure*

Although a common repertoire of functions was in principle made available to each compiler phase, any given phase used but a fraction of them, and single programs substantially fewer still. The only communication between compiler phases was through the external forms of tables, so a fresh environment had to be established for each phase. It was not desirable to provide the full set of functions for every phase because the space such code and data would occupy would be substantial. Balanced against this desire for modularity was an ease-of-use requirement which precluded having to do explicit initializations of empty tabular structures. These are undesirable because they are error-prone if unchecked, redundant if dynamically checked, and difficult to maintain, particularly to remove previously required but no longer needed function whose presence costs space and time.

These considerations led rather naturally to having many linked list structures in dynamically managed storage (PL/I BASED storage class). There was strong incentive to chain independently allocated table entries into a list in order to facilitate the insertion, deletion, and rearrangement functions required. Additionally, it was found useful to control these lists with a master list of existing tables, so that storage for table control is allocated only when a table is created and just those tables necessary are present.

The minimal initialization problem was thereby solved, since at the start of execution the list of tables is empty. Of course, this list must have some anchor in a fixed place, and a STATIC EXTERNAL variable whose name began with a reserved prefix was used for this purpose. This anchor was a POINTER, initialized to NULL, and was used only by those access function service routines (hidden from the application code) which locate, create, and destroy tables.

Each table was identified by a character string constant whose validity was checked at execution time. Those few functions which must identify a table (e.g., "INSERT_FIRST") typically used a character string constant parameter in the access function call. At program initialization all defined tables logically existed (with no entries), and inquiries about empty tables were legitimate, although the answers were inferred internally by the lack of a control block for that table. A table containing one or more entries had a control block, created by an attempt to insert an entry into an empty table. If a table name was not recognized, either because the access function subsystem implementation

had lagged or the name had been misspelled, a diagnostic was issued to the effect, "I don't know anything about this table." Table control blocks contained identification and status information such as the name of the table, the number of entries, and pointers to the first and last entries. Pointers to the table control blocks were not given out above the access function boundary; all functions supported on tables were done using the character-string name of the table.

Each entry within a table was chained to its immediate neighbors and to the control block for the table, so that the table to which any entry belonged could be quickly established. Each table entry had an "internal name," the value by which it was identified when an access function parameter was required. The most desirable state of affairs would have been to have a data type called "INTERNAL_NAME" whose use would be restricted solely to assignment, comparison for equality, and use as a parameter to the access functions. The data type POINTER has almost exactly these attributes when the access function entries are fully declared, and since when access functions are fully supported there is little need for pointers for other purposes, the safety is almost complete, by convention rather than by construction.

We made it a firm rule that POINTERs used as access function internal names could never be used as locators above the access function boundary, and descriptions of the internal list structures were never made available. As an informal coding standard, we agreed on the character @ (a letter in the PL/I lexical alphabet) to be the initial character of the identifiers for such pointers and not to use similar identifiers for other purposes. While this rule was not invariably followed, it turned out to be useful as a documentation convention.

We could have used the DEFAULT statement in PL/I to type identifiers beginning with @ as POINTER and thereby catch those we failed to declare; these otherwise defaulted to FLOAT data type and led to compiler diagnostics. However, we had agreed that all variables should be declared, and the language default almost inevitably guaranteed that the presence of such undeclared variables would not go long undetected. Similarly, we could have used the preprocessor to change the string "INTERNAL_NAME" to "POINTER," but there seemed little point to that, since we were not using pointers for any other purpose in our source code.

Because of the requirement that the implementation be in legal PL/I, it was not safe to provide a pointer directly to the specific structure for each table. This is because the structure mapping rules of PL/I are designed to ensure the integrity of substructures passed as parameters and not to define a storage mapping. As a result, slack bytes may be inserted in not easily predictable ways, and the language implies that the compiler may map structures in any way which permits substructures to be referenced as parameters. Thus, a structure containing at its head a descriptor identifying the format of the remainder of the structure is not entirely safe, for the descriptor itself might be assigned different offsets depending on the details of the remainder of the structure. What is indeed safe is to have a descriptor of a common format, which contains a pointer to a specifically formatted block, whose layout can be inferred from the descriptor block so that a correct access can be made.

The only pointers given out by the access functions and used above the access function interface were thus pointers to simple descriptor blocks, each of which contained four pointers: forward and backward chains to adjacent descriptor blocks within the table, a link to the control block for the table, and a link to the structure containing the specific data for that table entry. Because the interface to the access functions dealt with pointers to one type of structure only, checking could be much more thorough and the more error-prone pointer chaining operations were confined to small, regular sections of code.

In order to separate the maintenance activities of the access functions from the code which they serviced, the linkage was effected through external procedure calls. For a typical function, one would simply provide the name and give a single internal name parameter. For example, to obtain the @LABEL field of a particular text item whose internal name is held in the variable @TEXT, one would write

```
@LABEL (@TEXT)
```

where the access function entry is declared

```
DECLARE @LABEL ENTRY (POINTER)
    RETURNS (POINTER);
```

(What is returned is the internal name of an entry in the "LABEL" table or the NULL pointer in case the field is empty.)

- *Name translation*

Although variables of internal scope in PL/I can be as long as 32 characters, external variables are limited to seven characters. This restriction is contrary to the spirit desired for the access function approach, and so this problem was circumvented by using the PL/I preprocessor to translate the names of access functions from ordinary-appearing PL/I identifiers to seven-character identifiers consisting of a reserved prefix and an identification number for the function. The above function call as seen by the compiler would thus be

```
$S_0701 (@TEXT)
```

The name translation served two additional functions. First, it provided a convenient cataloging mechanism. Of the four-digit identification number, the first two designated the table and the last two the function within the table. Table-independent functions had the form "00xx" and in general the access functions for a particular table started with "yy01" and went up. The update functions for the corresponding fields started with "yy51." The designations "yy00" and "yy50" were useful in field maintenance when functions related to the same table were collected in the same file. The other useful function of external name translation is that a misspelled external name would not be translated, and if in excess of seven characters (as most were), would immediately give rise to a compiler diagnostic.

This style of translation suffered some disadvantages. Chief among them was that compiler diagnostics are phrased in terms of the tokens presented to the compiler *after* the name translation has taken place, and so diagnostics involving access functions appear in a foreign language. Second, the preprocessor declaration and assignment which effects the transformation requires one or two lines of listing, and thus several pages of irrelevant output per compilation. (We eventually post-processed the listings to remove this when the number of entry points and thus translations became in excess of seven pages of output per compilation.) Finally, the translation process is not selective and may translate internal variable names indiscriminately in cases in which a name coincides with an access function which is not used in a particular program—even when a particular table is not in use. Selective translation could alleviate these last two problems, but would require more overhead at the point at which we are trying to reduce it.

- *General subsystem functions*

General operations on all legal tables included a function telling how many entries were in the table, functions giving the internal names of the first and last entries, and procedures to insert a new row at either the front or the end of the table, returning the internal name of the new entry so created. In the case of empty tables, the @FIRST_ENTRY and @LAST_ENTRY functions returned NULL, and the INSERT_FIRST and INSERT_LAST procedures caused creation of the table control block and its chaining to the list of active tables as well as the creation of the entry itself.

General operations applicable to all table entries included functions giving the internal names of the next and previous entries, the character-string name of the table to which the entry belongs, procedures to insert a new entry either before or after a specified entry and return the internal name of the new entry so created, procedures to move a fixed number of entries starting at a particular point either before or after another specified point (with checking to ensure semantic correctness of the request—e.g., the target point must not be within the block to be moved), a procedure to delete a table entry, and a function to determine whether an alleged internal name in fact points to a legitimate (and undeleted) table entry.

The entries in many of our tables could be usefully identified with a string of characters, and so an "EXTERNAL_NAME" function was provided that was valid for most table types. This function proved particularly useful in the error diagnostic routines. In building some tables from their external representations, a cross-reference between external names was often used to indicate the corresponding internal cross-reference between internal names. A dictionary mechanism was therefore provided which enabled quick searches to be made within a specified table for a specified external name. This dictionary mechanism was also used to locate the table control blocks for those functions which needed to deal with them.

A common error handling and instrumentation mechanism was built into the core routines. Here the power of PL/I came most into play. Error messages were easy to formulate, format, and write both to output data sets and the virtual machine console. A trace back facility provided by the built-in procedure "PLIDUMP" was also used at the point of each error to give a dynamic trace of how the particular statement in which it occurred was reached.

- *Sets*

At an early stage of the design it was recognized that a provision for sets would greatly facilitate experimentation with proposed algorithms for register allocation. Elements were to be drawn broadly from table entries, but the nature of the domains might shift as the compiler development progressed. In any case, the size of the universe would in some instances be known only dynamically. We thus chose a linked list structure for set representation rather than a bit string. The flexibility this provided well repaid any inefficiencies which *might* have arisen. The sets implemented are actually sets of pointers, but in practice one thinks of them as sets of the objects pointed to. A set is then an unordered collection of objects, no two of which have the same *pointer* values. Set elements could thus be any internal name (or indeed any pointer, whether of access function origin or not), and an effort was made to obtain some speed in searching for a particular element in a set by keeping sublists and hashing the value sought to determine in which sublist it should appear.

However, the value sought was the address of a block of BASED storage allocated by the run-time environment of PL/I, and the hashed value had no logical connection whatever with anything else. It was indeed not even repeat-

able from one execution to the next in the same CMS virtual machine due to shifting boundaries of free storage as disk file directory entries came and went. This non-repeatability caused considerable consternation in debugging those procedures which operated on set elements in arbitrary order (i.e., the order provided by a particular access function on such sets). In the end, two versions of access functions on sets had to be provided: a fast, hashed implementation, and a slower but more regularly behaving version.

All the sets in use at a particular point were contained in a table much like any other; the general table-handling procedures were applicable to these table entries. The structure of each entry, rather than being tabular as in the other tables, was modified and examined by special primitives which gave the cardinality of the set, added an element to a set (if not already in it), told whether a particular element was in the set or not, and enumerated the set elements in an arbitrary but self-consistent way. Simple-minded intersection, union, and set difference operations were built from these primitives with a view to coding the functions directly if their performance became critical, which it never did.

Once the set access functions were available they were found highly useful in many situations other than the ones for which they were originally envisioned. The ability to represent an essentially arbitrary universe played a large part in this utility. Instrumentation within the access functions eventually revealed that while the size of the total available universe of internal names could be in the thousands, the average cardinality of sets actually used was somewhat less than two. It appears that in case of doubt sets were selected when table structures were specified in lieu of elementary data items, in anticipation of future algorithmic generalizations which never materialized. Hashing thus turned out to be neither necessary nor helpful.

- *Operational procedures*

We had an almost consistent pattern of naming conventions for access function entry points. "Almost" was simply due to an early failure to appreciate the value of establishing systematic procedures in completely defining external interfaces, and by the time the desirability of this became apparent it was not worth applying retroactively. We thus had several variants of a function meaning "number of operands" in different spellings and abbreviations. For obvious and short-named functions such as "@OPERATOR" and "@LABEL" this problem never arose. For functions with identical semantics and attributes in more than one table arising in the later stages of development, the problem was avoided by having general agreement on names when tables or table columns were first defined and then monitoring for duplications and near-duplications thereafter.

In PL/I it is usually necessary to provide rather complete descriptions of external entry points in programs which call functions. This was particularly so in the case of the access functions, which invariably required pointer parameters. Two patterns of usage emerged. In one, only those functions used in a particular program were placed in entry declarations. In the other, all the entry points for accessing a particular table were placed in a source library, and all would then be included in any program using any element from the subject table. While the latter method made coding, documentation, and maintenance easier, the PL/I compiler would not discard the unused entry points and this made the resulting text files larger and the linkage process longer. In extreme cases, single procedures which referenced a few entries in several tables could contain more than 255 external names and run afoul of a CMS loader restriction. Such problems were handled on a case-by-case basis.

Another difficulty with entry declarations included from source libraries involved function names shared among two or more tables. The compiler would not permit multiple, consistent declarations, so commonly occurring names had to be placed in yet another source library member. While the problems were not difficult to solve, it is not clear that the time required to get around them could not have been more profitably spent in providing more flexible coding tools.

The compiled access function code was placed in an object text library, and routine manual tools sufficed to maintain this library. However, a problem arose with the nature of the individual object files produced by each external PL/I compilation. An external procedure gives rise to at least three entries in the external symbol dictionary—a control section containing the code, a static data control section, and the entry point to the procedure itself. Procedures having alternate entry points, of course, have additional entries in the external symbol dictionary. In addition, a standard control section called PLISTART is included with each external compilation. It must be entered prior to PL/I compiled code being executed in order to set up (via library routines) the run-time environment of PL/I. This control section has two secondary entry points, making a total of six control section and entry points for a single external procedure. The CMS text library mechanism had a limit of 1000 symbols in the directory, and this would have limited us to a maximum of 166 external procedures. It was thus necessary to post-process the object decks for the access function procedures to remove the extraneous PLISTART control section. This process was made routinely mechanical and served a useful purpose for the compiler code as well.

- *Maintenance*

The addition of a new column to an existing table (and thus an additional field in each entry) was quite straightforward.

A source library element containing the structure declaration for the prototypical element of the table was augmented, the appropriate access and update functions were created from extremely stereotyped forms, and the entire set of access functions for that table was recompiled against the new data structure for that table. Certain key routines such as those which allocate and free storage had also to be recompiled to reflect the new data requirements for table entries. After this was done, the object library was completely recreated, so that internal consistency was always maintained. Programs not requiring the new fields ran perfectly well with old libraries, but when reloaded against the new library required more space for data at execution, although the behavior was otherwise exactly the same. There was no way in which an inconsistent set of functions could be linked together, since no partial link editing was done.

The addition of a new table was only slightly more complicated. The chief difference was that the structure for the data element had to be created and placed in the source library used by the access function code and the name of the table had to be made known to the routine which deals with the character string representation of table names. These global operations having been done, the columns could then be added in the same way as described above.

One thing we had not anticipated was that functions would become obsolete. Fortunately, the maintenance mechanism coped with that problem quite nicely. The first thing to be done with the function that was to be eliminated was to change the name translation process so that instead of being translated to the string "$S_xxyy," it would be translated to the string "DEFUNCT_ACCESS_FUNCTION," ensuring a compiler diagnostic when a program formerly using that function correctly was recompiled. To protect against previously compiled programs trying to execute that function, the serialized names were never reassigned. Rather, entries into a general diagnostic were made for each of these functions to deliver a rather rude message to the perpetrator. In view of the total intertwining of access functions with the functional code of the experimental compiler, it was somewhat surprising that these messages never had occasion to be produced.

- *Miscellaneous*

One of the biggest problems turned out to be how to deal with table entries which had been deleted, since pointers to them might still be outstanding. Freeing and re-allocating the storage would lead to all sorts of anomalies, since a pointer to an entry in some table could under unfortunate circumstances end up being a valid pointer to an entry in some other table, and diagnosing the faulty logic leading to this circumstance could be particularly difficult.

In the end, a safe approach on both sides of the access function interface emerged. On the compiler side, deletion of table entries was required generally in two circumstances. One was where the ordering of table entries was irrelevant, the set table being the best example of this. If a table entry were needed for working storage during some process and could then be freed, this was no problem. The other case involved tables such as the intermediate text, in which order was very important and in which cross-references existed, making logical deletion (as opposed to telling the access functions to reclaim a particular entry) a tricky process. In this case, the only procedure reliable in the long run involved marking text to be deleted during the complicated optimization processes and then passing over the text later to remove the unneeded items, keeping intact the logical relationships established above the access function interface.

Beneath the access function interface we finally settled on the "tombstone" approach [10], and although not implemented terribly elegantly, this certainly turned out to be the function required.

The access function subsystem had no way of knowing which pointers were still available above the interface, and in principle every element in every table was accessible anyway. Under-the-covers garbage collection was thus not possible, and the experimental compiler code was not always as scrupulous as it might have been about helping the storage management process. The other difficulty was that the entire space was managed by the PL/I-based storage allocation mechanism, and the storage seemed to be quite fragmented in some cases, leading to large working sets in our virtual machines.

Operational characteristics

- *Performance*

Work began on the fundamental subsystem routines several weeks in advance of coding the initial compiler components. The tables most fundamental to the compiler were thus implemented in time for the initial stages of program testing. The subsystem proved quite useful throughout a fairly lengthy development process which proceeded approximately top-down, and during this time performance issues did not arise.

Once the basic compiler functions were in place, experimentation began on alternative processing algorithms which required more tabular information and more accesses to the data generally, and ultimately overall performance became of some concern. Using various program measurement tools available to us, we found that around one third of the execution time was spent in the access function code. In order to assess how much this might be reduced, timing experi-

ments were undertaken on a typical function exemplified by the PL/I statement

@Q = @LABEL (@P);

This statement was executed in a tight loop with various recipients of control on the called side of the interface in order to determine where the time was spent. The times listed below represent virtual CPU times as measured from CMS on an unloaded IBM System/370 Model 168. The times are optimistic compared to those to be expected in practice, as the cache and translation lookaside buffer hit ratios will be unusually high in the tests.

Setting up parameters on calling side	3.0
Linkage overhead in called procedure	11.3
Function in called procedure	8.9
Total time	23.2 μs

In the case of such a simple function the linkage overhead is a substantial portion of the execution time required. A very few registers are required, but the contents of all are saved and restored, and unnecessary storage allocation operations are performed. To see the extent of these inefficiencies, this routine was very carefully coded in assembly language, and the following results obtained:

Setting up parameters on calling side	3.0
Linkage overhead in called procedure	0.8
Function in called procedure	2.2
Total time	6.0 μs

At best, then, one third of the execution time could be quartered, leading to an overall improvement of 25%. It is not likely that this figure could be attained in practice due to the difficulties in maintaining such large amounts of assembly language code.

The performance of the access functions was also restricted by the fact that the functions were always external procedures to the compiler programs. As a result, the code surrounding the calls to them was generated quite conservatively by the PL/I compiler and potential optimizations could not be realized. In particular, the retrieval function entries could be given the attribute REDUCIBLE, which means that successive calls with the same arguments produce the same results. (In the case of these procedures, calls with improper arguments also produced side effects, but insofar as the calling procedure is concerned these are not noticeable.) Because many of the functions represented cross-references among tables, access functional composition occurred rather frequently. If a subexpression within such a composition were subsequently required, this could not be achieved with a single call unless that were done into an explicitly declared temporary variable to be reused, as the compiler ignored the REDUCIBLE attribute, even in the simplest of cases.

Ultimately the biggest difficulty arising from the use of this implementation was the working set size in the virtual machine. The access functions themselves were fairly large (typically several hundred bytes each), and the nature of the development process made infeasible the physical separation of the few instructions which performed the function in the normal case and the larger amount of diagnostic code which was only called into play infrequently. So a fragmentation of functional code existed from the outset.

However, we eventually found that the working set size of the entire program was substantially larger than the number of pages occupied by all code, and it was therefore clear that logically related data areas were quite fragmented. On the one hand, this cannot be regarded as surprising since one of our strongest motivations for the access function subsystem was the freedom from this sort of consideration. On the other hand, this is a source of some concern because the basic design did not anticipate this difficulty and could not easily address it.

• *Diagnostic help*

Substantial design and coding effort had gone into the diagnostic facilities with the intention of providing a uniform framework for error reporting and recovery so that the incremental work involved in column and table augmentation would consist solely of adding function. This large effort was to be justified by spreading its cost over a very large number of low-probability events. This strategy was quite successful. The name translation mechanism discussed earlier was an unexpected part of this function, albeit not an entirely desirable one when it came to examining compiler listings. Table type checking was done on all functional references for two reasons. One was to ensure that the function requested was valid for the particular table in which the argument of the function appeared. The other was to discriminate between tables having identically named fields, since the actual storage offsets might vary. A violation produced a message of the broad form

YOU HAVE REQUESTED FUNCTION *x* OF ENTRY *y* IN THE *z* TABLE; THIS FUNCTION IS NOT DEFINED.

In the case of retrieval functions, arbitrary values were returned; for example, NULL for pointers, zero for integers, a string of asterisks for character strings, and the like. When an improper update was requested (which happened considerably less frequently), no change in the tables was made.

Within table entries it was possible to check internal consistency. For example, in our text tables we had a field indicating the number of operands in the particular text unit. The update and retrieval functions for the operands themselves had two arguments—the internal name of the entry

and the ordinal number of the operand. Attempts to update or retrieve operands numbered between one and the current value of the number field were legal; all others were reported as errors. We did not check consistency among entries in the same table, nor among entries in different tables, this being the job of the compiler code proper.

• *Reliability*

The subsystem turned out to be extremely reliable. The head start it enjoyed in the implementation schedule helped considerably in this. There was enough time to make a very careful design which placed the critical functions in a few key modules which could then be produced with great care. All list linking operations fell into this category, for example, as did the storage management for the entire collection of tables. These core routines were extensively checked by walkthrough-like techniques at both a pictorial level and with the code itself, and these routines never misbehaved. Furthermore, they were support routines for the access functions and many of them could only be called from a small number of places. There were thus many layers of protection for those functions which were critical to the smooth operation of the subsystem.

Since in the compiler code pointers were not required for any purpose except the access function interfaces, we obtained a large amount of protection from the PL/I language. Pointers can only be used to point, as parameters to subroutines, in simple assignments of the form "A = B;", in comparisons for equality, and with the BUILTIN pointer-valued function NULL. We did not need to use them to point, and the other legal uses of pointers covered exactly the amount of function required to make the access functions useful. A misuse typically arose from the failure to declare a variable as a pointer. Since we agreed that most pointers should begin with the symbol "@," such undeclared variables would default to floating-point and some rather anguished diagnostics would ensue from the compiler, since pointer to floating-point data-type and *vice versa* are two of the few conversion combinations not automatically coerced in PL/I.

The prototypes for the access functions were very carefully designed to make the production of new functions and tables as fast as possible. After some trial and error, this led to a set of prototypes which were used for all but the very special set functions. When complete, the older functions were retrofitted to the prototypes as requests came in to add columns to the respective tables, and eventually all functions were implemented from the same set of prototypes. This kept the number of logical interfaces constant while the number of actual interfaces was expanding, and reliability of the ensemble was further enhanced.

A few misleading error diagnostics were produced in the early stages before that mechanism was also unified, and feedback from early usage soon helped correct these situations.

The source library by which the names were translated and the text library containing the access function code were maintained on a slowly changing, project-owned disk. When new functions were requested, the name translation entries were added immediately, enabling compilation of the compiler code needing these functions. (The name translation is also used in compiling the access functions themselves, so this is a necessary first step in any event.) The functions, when completely compiled along with any necessary additions to the core routines, were then placed in a new text library created afresh. Then the compiler programs requiring the new functions could be loaded and executed. This turned out to be a very satisfactory working arrangement.

Influence on the experimental compiler

Because the nature of the experiment was to demonstrate feasibility of ideas and algorithms, little emphasis was placed on the logical structure of the data above the access function interface. Many concerns about representation disappeared with the introduction of sets, and more effort was spent ensuring that needed information would be available than to ask whether it was made available in an efficiently accessed form.

The nature of the access function implementation discouraged experimentation with data structures and organizations, in some cases unnecessarily. It certainly made any direct comparisons of alternative data structures difficult, and this kind of experiment was never attempted. If sufficient thought and experience led to the conclusion that some information should be differently structured, then such restructuring was quite feasible. However, this did not happen as often as it might, perhaps because the perceived overhead of doing so was greater than the actual work involved.

As a result, some of the compiler algorithms became rather clumsy and certainly more convoluted than necessary, because it was always easier to patch up problems with a few more lines of code than to request a data organization change. In retrospect, it seems we should have been more sensitive to this problem, because the inflexibility which resulted made the experimental compiler much larger and slower than we suspect it had to be.

Summary of experience

The experimental compiler was under development for different purposes over a period of four years. The portions using the access function subsystem grew to four phases and 40 000 lines of PL/I code. The access function subsystem

comprised 421 entries embodied in about 22 000 lines of PL/I. During the two years between the time the compiler first functioned and the time the experiment ended, substantial function was added to support more source language and to produce better object code for several different machine targets. The utility of the access functions extended throughout this period, and we found their help as useful at the end as at the beginning. Their existence certainly extended the life of the code, all of which was written by people with no previous experience in coding such a large system in PL/I.

We had considered authorization checking as passive protection against the unwanted alteration of data. This turned out not to be necessary. The validity checking was quite sufficient for this purpose.

Syntactic errors such as the misspelling of function names and semantic errors such as requesting a function not defined on a particular table were quickly caught in a fashion which enabled speedy diagnosis and correction.

The objectives we had set to justify the labor involved in this approach were satisfied for the most part. The long-term drawbacks were the perceived enshrinement of data structures and the storage fragmentation which arose from lack of foresight.

The benefits we expected proved more valuable than had been anticipated. This is because we had not taken sufficient account of the quick and direct feedback provided by this subsystem in an interactive development environment. Each of us was working in a CMS virtual machine under the VM/370 operating system, and a great deal of this work was done on video display terminals. As a result, errors could be corrected almost immediately and the compilation or program test run again. (This notion seems routine today, but was novel to us then.)

The VM/370 environment also obviated the need for an automated maintenance system for the functions. A collection of simple procedures and code prototypes reduced the manual operation requirements to just those necessary to define the new structures and functions being introduced. The CMS editor used on the video terminal made it possible to produce the actual code by filling in a few blanks in the carefully composed prototypes. When the computer was lightly to moderately loaded, a new table could be introduced into the system in about ten minutes. This job comprised making the new PL/I structure declarations for table elements, inserting them into a source library used by the access functions, and changing and compiling the storage management routines to allow them to allocate and free instances of the new structure. From there, adding functions to the new or already existing tables could be done in about one minute per function, including compilation time.

At the end of the experiment we knew how to generate the access functions mechanically from simple descriptors of a table entry, but it was never worth a complete re-implementation to do so. Other groups subsequently made this sort of refinement to the methodology for their own purposes [11, 12].

The major problems over the life of the experiment were all essentially annoyances. The first concerns the nature of the PL/I language. While extension via functional notation is quite simple because the notion of function is quite strong in the language, updating the entries is not as convenient, since the "pseudo-variables" of PL/I are confined to those which are BUILTIN. Since we were referencing rather than updating most of the time, this was a minor difficulty, but those procedures which had to build tables were found cumbersome to code. This asymmetry between function and pseudo-variable was not considered serious enough to warrant a frontal attack, e.g., through the macro pre-processor.

The second problem was the working set size. While the compiler programs and access functions were easy to instrument for execution time performance, determination of static storage requirements was rather more difficult, and dynamic determination of storage usage patterns was extremely expensive in both human and computer time. Since CMS must always run in a virtual machine, perhaps more sympathetic help could be expected of it.

Finally, due to a combination of PL/I and CMS restrictions, we were unable to read and write directly the internal structures created by the access function subsystem. This cost us machine time in our testing in having to build up the same tables from their external representations over and over. While this problem could also have been circumvented, this wasted machine time turned out to be less costly than the human time which would have been required to redesign the access function internals.

Acknowledgments

The notion that access functions with these characteristics belonged in the experimental compiler was due to Hans Schlaeppi, the project manager. Aspi Wadia helped with an early design for the core procedures which turned out to be invaluable because we made all the list processing mistakes once and then threw it all away to build the final version. The subsystem was then given a thorough workout by Joonki Kim, Hans Schlaeppi, C. J. Tan, Ray Villani, and Aspi Wadia. Their timely criticisms and comments kept the design on the path of reasonableness and usefulness. I am grateful to Ray Villani, Aspi Wadia, and Hank Warren for helpful comments on early drafts of this paper and to the anonymous referees for valuable suggestions.

References

1. B. H. Liskov, A. Snyder, R. R. Atkinson, and J. C. Schaffert, "Abstraction Mechanisms in CLU," *Commun. ACM* **20,** 564–576 (1977).
2. B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. L. Popek, "Report on the Programming Language Euclid," *Sigplan Notices* **12** (February 1977).
3. N. Wirth, "Modula: A Language for Modular Multiprogramming," *Software Pract. Exper.* **7,** 3–35 (1977).
4. J. D. Ichbiah, J. C. Heliard, O. Roubine, J. G. P. Barnes, B. Krieg-Brueckner, and B. A. Wichmann, "Rationale for the Design of the Ada Programming Language," *Sigplan Notices* **12** (June 1979).
5. *Abstract Software Specifications*, D. Bjorner, Ed., Springer-Verlag New York, Inc., New York, 1979.
6. *Proceedings of the Workshop on Data Abstraction, Databases, and Conceptual Modeling*, M. L. Brodic and S. N. Zilles, Eds., Association for Computing Machinery, New York, 1979.
7. B. Kutzler and F. Lichtenberger, *Bibliography on Abstract Data Types*, Springer-Verlag New York, Inc., New York, 1983.
8. M. Shaw, "The Impact of Abstraction Concerns on Modern Programming Languages," *Proc. IEEE* **68,** 1119–1130 (1980).
9. W. A. Wulf, "Abstract Data Types: A Retrospective and Prospective View," *Proceedings of the 9th Symposium on the Mathematical Foundations of Computer Science*, Springer-Verlag New York, Inc., New York, 1980, pp. 94–112.
10. D. B. Lomet, "Scheme for Invalidating References to Freed Storage," *IBM J. Res. Develop.* **19,** 26–35 (1975).
11. F. E. Allen, J. L. Carter, J. Fabri, J. Ferrante, W. H. Harrison, P. G. Loewner, and L. H. Trevillyan, "The Experimental Compiling System," *IBM J. Res. Develop.* **24,** 695–715 (1980).
12. John A. Darringer, William H. Joyner, Jr., C. Leonard Berman, and Louise Trevillyan, "Logic Synthesis Through Local Transformations," *IBM J. Res. Develop.* **25,** 272–280 (1981).

Received February 17, 1983; revised September 8, 1983

Frederic N. Ris *IBM Research Division, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Ris is senior manager of computation-intensive systems in the Computer Science Department at the Thomas J. Watson Research Center. Dr. Ris received a B.A. from Harvard College in chemistry and physics and a Ph.D. in mathematics from Oxford University, England. He joined the IBM Research Division in 1972 to participate in a project concerned with machine-dependent optimization for the generation of microcode from high-level languages. In 1977 he became manager of a project to develop system software tools and prototype applications for a novel digital signal processor architecture which he helped generalize and for which he was awarded an IBM Outstanding Innovation Award. He has served as technical assistant to the director of computer sciences in Research and has continued work begun at Oxford on computer arithmetic. In 1980–1981, Dr. Ris served as technical assistant to the IBM Chief Scientist and executive secretary to the IBM Science Advisory Committee at Corporate Headquarters in Armonk, New York.