Aspects of Implementing CLU

Russell R. Atkinson
Barbara H. Liskov
Robert W. Scheifler

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Linguistic mechanisms used in CLU to support 1) structured exception handling, 2) iteration over abstract objects, and 3) parameterized abstractions are briefly reviewed, and methods of realizing these mechanisms are described. The mechanisms discussed support features that are likely to be included in other programming languages, and the implementation methods should be applicable to a wide range of languages.

Key words: CLU; exception handling; iterators; parameterized modules; programming language implementation methods.

1. Introduction

CLU [1, 2] is a programming language/system that provides linguistic mechanisms supporting procedural, data, and control abstractions, and structured exception handling. In implementing CLU, methods of realizing these linguistic mechanisms were needed. This paper describes a few of these implementation methods. An implementation using these methods has been running since the fall of 1977.

The methods we describe are ones that we feel are of general interest because the features they support are likely to be included in other programming languages. In Section 2 we discuss how the CLU exception handling mechanism is realized. In Section 3 we describe the implementation of iterators. Finally, Section 4 contains a discussion of the implementation of parameterized modules. More complete descriptions of exception handling and iterators can be found in [3] and [4], respectively.

This work was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-75-C-0661.

2. Exception Handling

In CLU, a routine (procedure or iterator) can terminate in one of a number of conditions. A routine terminates in the *normal* condition by executing a **return** statement; termination in an *exceptional* condition is accomplished by executing a **signal** statement. In each case, result objects may be returned; the result objects may differ in number and type in the different cases.

The information about the ways in which a routine may terminate must be included in its heading. For example, the procedure performing integer division has the following heading:

```
div = proc (x, y: int) returns (int) signals (zero_divide)
```

which indicates that *div* may terminate normally by returning a single integer (the quotient of the two input arguments), or exceptionally by signalling *zero_divide* (which indicates that the second argument was zero) and returning no results. In addition to the named exceptions, every routine can terminate in the special *failure* condition, with a string as the result.

The exceptions signalled by a routine must be caught and handled by its immediate caller. Handlers are associated with invocations statically, and are placed in CLU programs by means of the **except** statement, which has the form:

statement **except** *handler list* **end**

This statement has the following interpretation. When a routine activation terminates by signalling an exception, the corresponding invocation (the text of the call) is said to *raise* that exception. If, during execution of the *statement*, an invocation raises an exception, control immediately transfers to the closest applicable handler; i.e., the closest handler for the

exception that is attached to a statement containing the invocation. When execution of the handler completes, control passes to the statement following the one to which the handler is attached. Thus if the closest handler is in the *handler list*, the statement following the **except** statement is executed next. If execution of the *statement* completes without raising an exception, the attached handlers are not executed.

Each handler in the *handler list* names one or more exceptions to be handled, followed by a list of statements (called the *handler body*) describing what to do. Several forms are available for handlers. For example, the objects provided by the signaller may be discarded if desired, and a special **others** form may appear last in the list to handle all exceptions not handled by other handlers in the list.

The example below illustrates the association of handlers with exceptions:

```
begin  % start of inner block
   S1 except when zero: S2
             end
   ...
end  % end of inner block
except when zero: S3
       others: S4
       end
```

If *zero* is raised by an invocation in *S1*, it will be handled by *S2*, not *S3*. However, if *zero* is raised by an invocation in *S2*, it will be handled by *S3*. All other exceptions raised in *S1* and *S2* will be handled by *S4*.

Sometimes there is nothing useful that a calling routine can do when an exception is raised by some invocation, so no handler is provided. In this case, the uncaught exception automatically turns into a *failure* exception of the calling routine with the string result:

"unhandled exception: *name*"

where *name* is the name of the unhandled exception.

2.1. Implementation

There are several possible methods of implementing the exception handling mechanism. As usual, tradeoffs must be made between efficiency of space and time. We believe the following are appropriate criteria for an implementation:

1. normal case execution efficiency should not be impaired at all.
2. exceptions should be handled reasonably quickly, but not necessarily as fast as possible.
3. use of space should be reasonably efficient.

The tradeoff to be made is the speed with which exceptions are handled versus the space required for code and data used to locate handlers.

Signalling an exception involves the following actions:

1. discarding the activation record of the signalling activation (but saving the result objects associated with the exception).
2. locating the appropriate handler in the calling routine.
3. adjusting the caller's activation record to reflect any terminations of expressions and statements containing the invocation.
4. copying the result objects into the caller's activation record.
5. transferring control to the handler.

Actions (3) and (5) are equivalent to a *goto* from the invocation to the handler. Actions (1) and (4) are similar to those occurring in the normal termination of a routine. Because the association between invocations and handlers is static, the compiler can provide the information needed to perform actions (2) and (3). Below we sketch two methods of providing this information; these methods differ considerably in their performance characteristics.

The first method, called the *branch table method*, is to follow each invocation with a branch table containing one entry for each exception that can be raised by the invocation. Each entry contains the location of a handler for the corresponding exception. The invocation of a routine whose heading lists n exceptions will have a branch table of $n + 1$ entries; the first n entries correspond to the exceptions listed in the heading, while the last entry is for *failure*.

Using this method, **return** and **signal** statements are easy to implement: **return** transfers control to the location following the branch table, while **signal** transfers control to the location stored in the branch table entry for the exception being signalled. The information needed to adjust the caller's activation record could be stored with the handler, as could information about whether to discard the returned objects; for example, this information could be placed just before the first instruction of the handler. Figure 1 shows the code skeleton for an invocation using this method.

To invoke p = **proc ( ) returns ( ) signals** (e1, e2):

```
            call  p
            e1_addr          ; branch table
            e2_addr
            failure_addr
            ...              ; normal return here
- - - - - - - - - - - - - - - -
            size1            ; new activation record size
            ...              ; other info about the handler
e1_addr:    ...              ; code for e1 handler
            ...
```

Figure 1. Code skeleton using the branch table method

The branch table method provides for efficient signalling of exceptions, but at a considerable cost in space, since every invocation must be followed by a branch table (all invocations may at least signal *failure*). A second method, the *handler table method*, is the one used in the current CLU implementation. The method trades off some speed for space, and was designed under the assumption that there are many

fewer handlers than invocations, which is consistent with our experience in using the mechanism.

The handler table method works as follows. Rather than build a branch table per invocation, the compiler builds a single table for each routine. This table contains an entry for each handler in the routine. An entry contains the following information:

1. a list of the exceptions handled by the handler (a null list can be used to indicate an others handler).
2. a pair of values defining the *scope* of the handler, that is, the object code corresponding to the statement to which the handler is attached.
3. the location of the code of the handler.
4. the new activation record size.
5. an indication of whether the result objects are used in the handler.

The exception list and scope together permit candidate handlers to be located: only an invocation occurring within the scope and raising an exception named in the exception list can possibly be handled by the handler (for an others handler, only the scope matters).

A return statement is implemented just as it would be in a language without exception handling. A signal statement requires searching the handler table to find entries for candidate handlers; if several candidates exist, the one with the smallest scope is selected. Placing the entries in the table in the linear order in which the corresponding handlers appear in the source text guarantees that the first candidate found is the handler to use. Unhandled exceptions can be recognized either by the absence of candidates or by storing one additional entry at the end of the handler table for this case.

3. Iterators

Iterators are a type of control abstraction that permit iteration over a collection of items (such as the elements of a set or an array) without knowledge of how the items are obtained from the collection. An iterator produces the items in the collection one at a time; it can only be invoked by a for statement, which uses the items in performing some computation. Details of how the items are selected from the collection are local to the iterator; the for statement simply uses the items, without knowledge of how they are produced.

The heading of an iterator states the types of arguments that the iterator requires and the types of objects making up each item (an item consists of zero or more objects). For example, CLU arrays provide an *indexes* iterator which produces the sequence of integers that are legal indexes into the given array. The heading of this iterator is

```
indexes = iter (a: array[t]) yields (int)
```

where *t* stands for the type of element in array *a*.

An iterator produces an item by executing a yield statement; the iterator is then suspended and its state is retained. When another item is needed the iterator is resumed with the saved state, from which it continues and possibly produces additional items. Thus an iterator is a coroutine [5, 6]. However, the use of iterators is limited to permit efficient implementation, as discussed further below.

The CLU for statement has the form:

```
for variable-or-decl-list in invocation do
    body-of-statements
    end
```

The iterator is initially called in the *invocation*. Each time an item is yielded, the objects in the item are assigned to the variables in the *variable-or-decl-list*. Then the *body-of-statements* is executed to make use of the objects produced. At the end of the *body-of-statements* the iterator is resumed to produce another item. When the iterator terminates (because the sequence is exhausted), the for statement also terminates. The variables in the *variable-or-decl-list* can be global to the for statement (a *variable-list*), or local to the for statement (a *decl-list*). These variables are not shared with the iterator, so assignments to them in the for loop body cannot affect the iterator's behavior. Therefore, such assignments are not forbidden.

It is also possible for a for statement to terminate both itself and the invoked iterator. The break statement terminates the smallest enclosing for statement (they can be nested) and the iterator it invoked. An exception raised by an invocation in the for loop body but handled outside the for statement will terminate both the for statement and the iterator. Execution of a return or signal statement terminates both the enclosing routine and all iterators it has invoked.

3.1. Implementation

The use of iterators is restricted in two ways: iterators are invoked only by for statements, and each for statement invokes exactly one iterator. These restrictions insure that active iterator invocations are always nested, which in turn means that iterators can be implemented using a single stack.

The implementation of iterators is similar to that of procedures. Both share a single stack for arguments, linkage information, variables and temporaries. In the following discussion we assume that the reader is familiar with stack frames (also called activation records). Also, although it does not really affect the implementation, it may help to know that CLU modules are not nested in one another and have no free variables.

Each stack frame contains the following linkage information:

1. the return address in the calling routine.
2. the routine base address for the current routine (used, e.g., for finding literals).
3. the return link, which points to the base of the stack frame for the calling routine. There is a single call chain, which consists of stack frames linked via return links.
4. the resume link, which is used to chain together the

information necessary to resume suspended iterators invoked by the current routine (null if no suspended iterators). An iterator chain consists of stack frames linked via resume links. Each routine may have a separate iterator chain.

The arguments to the routine are directly above the frame's linkage information (assuming the stack grows down), and the local variables are directly below (all local variables are allocated together at routine entry, rather than at block entry). Below the local variables are temporaries for invocations (the temporaries become arguments to other routines). Figure 2 shows a stack frame.

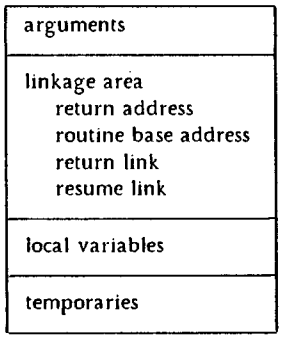

Figure 2. A stack frame (assuming the stack grows down)

Calling an iterator is identical to calling a procedure: a new stack frame is created, and the return address, routine base address, and return link are set. The resume link is initially null, since there are no suspended iterators. Returning from an iterator is identical to a normal procedure return: the iterator's frame is removed from the stack and control transfers to the return address.

Yielding is somewhat like invoking the for loop body as a normal procedure. When an iterator yields an item, the frame of the iterator remains on the stack, and a special frame, called a *resume frame*, is added to the stack to hold information about how to resume the iterator. This resume frame is an abbreviated version of a normal frame, containing only linkage information. The return address contains the location where control will go when the iterator is resumed and the return link points to the frame for the iterator; the routine base address is null to indicate that this is a resume frame. The resume frame is added to the iterator chain of the routine that called the iterator: the resume link for the calling routine's frame is set to point to the resume frame, and the resume link for the resume frame is set to the previous contents of the resume link for the calling routine's frame. Finally, control transfers to one beyond the return address given to the iterator (the actual return address is used for returning and contains a branch around the loop body).

Resuming an iterator is similar to returning from a procedure. When a for loop body terminates, the iterator to be resumed is the first one on the iterator chain for the current frame. Its resume frame is removed from the chain: the resume link of the current frame is set to the resume link of the resume frame. The information in the resume frame is used to locate the iterator's stack frame and resume point. Then the resume frame is removed from the stack. At this point, the stack pointer has the same value as it had when the iterator last yielded an item, so the stack can grow as necessary during the iterator's execution.

Executing a return or signal statement in the middle of a for loop is as efficient as a normal return or signal. The stack space for the suspended iterators is reclaimed when the stack space for the current frame is reclaimed. Early termination of the for loop in any other way (e.g., by a break) requires a few instructions to reclaim the stack space for the appropriate suspended iterators.

Consider a routine *P* that contains a nested for loop of the form:

```
for ... in Iter1(...) do
    for ... in Iter2(...) do
        ...
    end
end
```

Assume also that iterator *Iter2* contains a for loop of the form:

```
for ... in Iter3(...) do
    ...
end
```

Figure 3a shows the stack as it appears whenever the execution of *P* is inside the outer loop but not in the inner one. *Iter2* has not yet been invoked (or its previous invocation has terminated). At this point, *P*'s iterator chain contains a single element: RF1, the resume frame for *Iter1*.

In figure 3b, the stack is shown as it appears whenever execution is inside *P*'s inner for loop. Now *P*'s iterator chain contains two entries: RF2, the resume frame for *Iter2*, and RF1, the resume frame for *Iter1*. RF2 appears first on the chain, so it is easily found when the inner loop body terminates. The iterator chain for *Iter2* contains a single entry: RF3, the resume frame for *Iter3*.

Iterators are inexpensive to implement. Yielding is similar to calling a routine (a frame is created), and resuming is similar to returning, so the cost of using an iterator is roughly equivalent to the cost of a procedure call for each execution of the loop body. Even this cost can be eliminated by doing inline substitution: the iterator body is substituted (with minor changes) for the for loop control, and the code of the for loop body is substituted for occurrences of the yield statement. If the iterator contains more than one yield statement, possible code duplication can be avoided by treating the for loop body as an internal procedure.[1]

1. Invoking this sort of procedure involves remembering only the return point; no arguments are passed and no context switch need be done [7, 8].

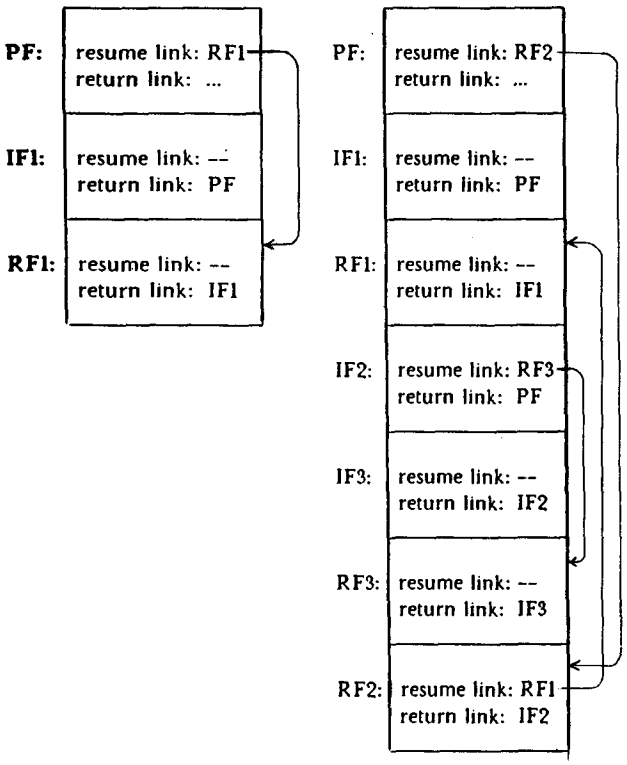

Figure 3a Figure 3b

RF*n*: resume frame for Iter*n*
IF*n*: normal frame for Iter*n*
PF: normal frame for procedure P

Figure 3. Examples of iterator chains

4. Parameterized Modules

In CLU, procedures, iterators, and clusters can all be *parameterized*. Parameterization provides the ability to define a class of related abstractions by means of a single module. Parameters are limited to just a few types, including integers, strings, and types. The most interesting and useful of these are the type parameters: objects in CLU can grow and shrink dynamically, so size parameters are not needed.

When a module is parameterized by a type parameter, this implies that the module was written without knowledge of what the actual parameter type would be. Nevertheless, if the module is to do anything with objects of the parameter type, certain operations must be provided by any actual type. Information about required operations is described in a where clause, which is part of the heading of a parameterized module. For example,

```
set = cluster [t: type] is create, insert, delete, elements
      where t has equal: proctype (t, t) returns (bool)
```

is the heading of a parameterized cluster defining a generalized set abstraction. Sets of many different element types can be obtained from this cluster, but the where clause states that the element type is constrained to provide an *equal* operation.

As a second example, the following parameterized procedure defines a class of summing functions for collections (such as sets and arrays) of integers:

```
sum = proc [struc: type] (s: struc) returns (int)
      where struc has
            elements: itertype (struc) yields (int)
    x: int := 0
    for elt: int in struc$elements(s) do
        x := x + elt
        end
    return (x)
    end sum
```

The where clause constrains the legal actual type parameters to those having an *elements* iterator of the appropriate type.

To use a parameterized module, actual values for the parameters must be provided, using the general form

```
module_name [ parameter_values ]
```

Parameter values must be computable at compile-time. Providing actual parameters selects one abstraction out of the class of related abstractions defined by the parameterized module; since the values are known at compile-time, the compiler can do the selection and can check that the where clause restrictions are satisfied. The result of the selection, in the case of a parameterized cluster, is a type, which can then be used in declarations and operation names; in the case of parameterized procedures or iterators, a procedure or iterator is obtained, which is then available for invocation. For example, *sum*[*set*[int]] is a use of the two abstractions shown above, and is legal because int provides an *equal* operation and *set*[int] provides an *elements* iterator.

4.1. Implementation

There are a number of basic schemes for implementing parameterized modules. These schemes can be characterized by the time at which the binding of actual parameter values takes place. The possible times include compile time, load time (after compilation but prior to execution), and run time (either at the first use of each distinct set of parameter values, or at every use). The result of binding parameters is called an *instantiation*.

In a compile-time binding scheme, the compiler produces a distinct object module for each distinct set of parameter values; each use of a formal parameter in the source text is replaced by the corresponding actual parameter, and then the resulting text is compiled to obtain the instantiation. In the load-time and run-time schemes, a parameterized abstraction is compiled into a single, parameterized object module; this module is later instantiated by supplying actual values for the parameters.

The compile-time scheme is similar to macro processing, and has many of the associated advantages and disadvantages. Its primary advantage results from the greater context that is available to the compiler when compiling any particular instantiation of a parameterized abstraction. This increased context allows the generation of more time-efficient object modules, both because of the greater opportunities for optimization and because run-time binding is avoided. The primary disadvantages of this scheme are the increased number of compilations performed and the increased amount of space needed to store the object modules.

In the load-time and run-time schemes, binding is performed on object modules. The binding does not require that a new copy of an object module be created for each set of parameter values; rather, the code of the module and most of its local data can be made independent of the particular parameter values, and thus can be shared by the various instantiations.

There are two possible run-time schemes. In the first, the binding of parameters takes place each time a parameterized object module is invoked. The parameter values are passed to the object module as extra, hidden arguments, and are referred to by the object module just like the normal, explicit arguments. In the second scheme, which is the one used in the current CLU implementation, a new object module is created once for each distinct set of parameter values; the binding occurs at the first use during execution.[2] The new object module is created by building a new structure containing the parameter-dependent data; the code of the module and its parameter-independent data are shared by the various instantiations.

Compile-time and load-time schemes all require that every possible set of parameter values supplied to an abstraction be determined before execution begins. In CLU, the possible parameter values are restricted to "compile-time computable" constants. However, despite this restriction, it is possible to implement recursive parameterized abstractions that use an unbounded number of distinct parameter values, as the following perfectly legal module (inspired by [9]) demonstrates:

```
agen = proc [t: type] (n: int) returns (any)
       if n <= 0
         then return (array[t]$new ())
         else return (agen[array[t]] (n - 1))
         end
       end agen
```

An invocation *agen*[*T*](*n*), where *T* is an arbitrary type, eventually produces a new array. The important characteristic of *agen*, however, is that *agen* calls itself recursively with a parameter array[*t*] that is distinct from the original parameter *t*; in fact, it is distinct from any previous parameter to *agen* within a single recursive chain of calls. For any positive *n*, an invocation of one instantiation of *agen* will use *n* distinct additional instantiations of *agen*. For example, the invocation *agen*[int](3) will result in 3 recursive instantiations of *agen*:

```
agen [array [int]] (2)
agen [array [array [int]]] (1)
agen [array [array [array [int]]]] (0)
```

Thus there exist finite CLU programs that use at run-time an unbounded number of instantiations of parameterized abstractions. To handle such programs, it is therefore necessary to support the dynamic instantiation of parameterized abstractions at run-time. For a compile-time scheme to be correct, one must recognize modules such as *agen* and either consider them to be illegal, or provide some means for implementing them that avoids compiling an infinite number of object modules.

As was mentioned above, the current CLU implementation utilizes a run-time scheme wherein a new object module is created once for each distinct set of parameter values. Since in the implementation there is no single object module for a cluster as a whole, but rather individual object modules for each cluster operation, the following (somewhat simplistic) description focuses on the representation of routines. Types are represented, by objects called *type descriptors*; however, type descriptors are used primarily in various forms of identification, and their internal format is not of particular importance here.

The implementation makes use of two types of objects, *call blocks* and *entry blocks*. A call block is a description of a routine to be invoked, and contains a type descriptor for the data type, if the routine is a cluster operation, the routine name, and the actual parameters for the routine.[3] An entry block represents an invocable entity (i.e., a non-parameterized routine or an instantiation of a parameterized routine); it contains references to constituent objects containing the code for the routine, the parameter-independent data, and the parameter-dependent data. The parameter-independent data consists of literal values, such as real numbers and strings, and call blocks for invoked routines that are not dependent on the parameters. There is parameter-dependent data only in entry blocks for instantiations; this data consists of the actual parameters and call blocks for invoked routines that depend on those parameters.

For example, figure 4 shows the entry block for the instantiation *sum*[*set*[int]]. This entry block refers to one parameter-independent call block, for int$*add*, and one parameter-dependent call block, for *set*[int]$*elements*. Notice that in the call block for *set*[int]$*elements* there are no routine parameters; this is because *elements* has no parameters besides those of its containing cluster. A call block for *sum*[*set*[int]] is shown in figure 5. Note that here there is a routine parameter, but no type descriptor, since *sum* is not an operation of a cluster.

2. However, one can run through storage looking for uses of parameterized modules and force binding to take place before execution.

3. Individual cluster operations can have parameters in addition to those for the entire cluster.

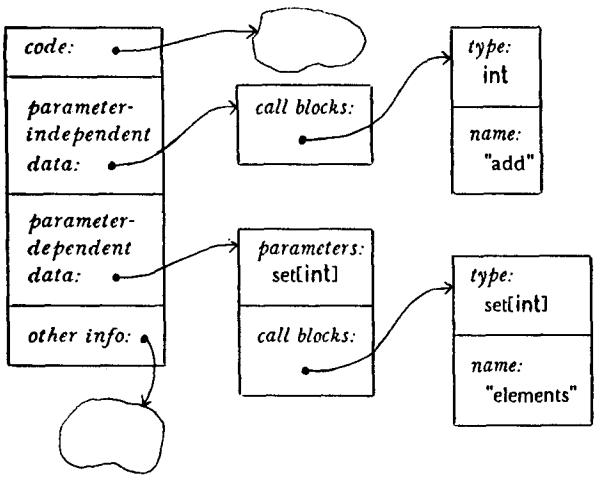

Figure 4. Entry block for *sum*[*set*[int]]

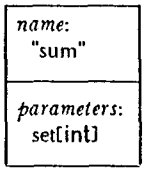

Figure 5. Call block for *sum*[*set*[int]]

The uninstantiated form of a parameterized routine is also represented by an entry block, to be used as a template when building instantiations. In the parameter-dependent data of this entry block, each would-be reference to the *ith* actual parameter is instead a reference to a dummy descriptor for "the *ith* parameter". For example, the template for *sum* looks like figure 4, except that references to *set*[int] are replaced by references to "the first parameter".

Whenever an attempt is made to invoke a routine through a call block, a dynamic linker intervenes. If the entry block for the specified routine already exists, the call block is replaced by that entry block, thus *snapping* the link. If the entry block does not yet exist, i.e., a parameterized routine is being instantiated with a new set of parameters, a new entry block must first be created from the template entry block for the routine. The new entry block shares the code and the parameter-independent data with the template (and all other instantiations), but has a completely new copy of the parameter-dependent data in which every reference to a dummy descriptor for "the *ith* parameter" is replaced by a reference to the corresponding actual parameter.

It is important to realize that instantiation merely involves substituting actual parameters into the parameter-dependent data template; no attempt is made to simultaneously snap the call blocks in the resulting data. One reason for this is that attempts to instantiate certain routines (such as *agen* above) would cause an infinite number of subsidiary instantiations. A second reason is that some (possibly many) of the call blocks may never be used, so snapping them is a waste of time. For example, code to handle potential, but unexpected, exceptions may never be executed.

The above description omits a number of details that are largely related to aspects of performance. For example, the parameter-dependent data in an entry block is actually separated into two parts: data dependent solely on cluster parameters, and data dependent on routine parameters (and perhaps also on cluster parameters); in this way, all operations of a parameterized type can share that data dependent on just the cluster parameters, while those (rare) operations that are additionally parameterized have separate, additional data dependent on those parameters. Although these details are important to the actual implementation, they do not fundamentally alter the description just given, and so will not be pursued here.

References

[1] Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C. Abstraction Mechanisms in CLU. *Comm. ACM 20, 8* (Aug 1977), 564-576.

[2] Liskov, B., Moss, E., Schaffert, C., Scheifler, B., and Snyder, A. CLU Reference Manual, *Computation Structures Group Memo 161.* M.I.T., Laboratory for Computer Science, Cambridge, MA (July 1978).

[3] Liskov, B., and Snyder, A. Structured Exception Handling, *Computation Structures Group Memo 155.* M.I.T., Laboratory for Computer Science, Cambridge, MA (Dec 1977).

[4] Atkinson, A., and Liskov, B. Iteration Over Abstract Objects in CLU, *Computation Structures Group Memo 167.* M.I.T., Laboratory for Computer Science, Cambridge, MA (forthcoming).

[5] Dahl, O.-J., and Hoare, C.A.R. Hierarchical Program Structures. *Structured Programming*, Academic Press, London (1972).

[6] Conway, M.E. Design of a Separable Transition-Diagram Compiler. *Comm. ACM 6, 7* (July 1963), 396-408.

[7] Allen, F.E., and Cocke, J. A Catalogue of Optimizing Transformations, *RC 3548.* IBM Thomas J. Watson Research Center, Yorktown Heights, NY (Feb 1975).

[8] Geschke, C.M. *Global Program Optimizations.* Ph.D. Thesis, Computer Science Dept., Carnegie-Mellon University, Pittsburgh PA (Oct 1972).

[9] Gries, D., and Gehani, N. Some Ideas on Data Types in High-Level Languages. *Comm. ACM 20, 6* (June 19[illegible], 414-420.