Request Functions for the C Language

Roger Burkhart
Deere & Company
John Deere Road
Moline, IL 61265
roger@ci.deere.com

Workshop on Object-Oriented Reflection and Metalevel Architectures
OOPSLA '93

ABSTRACT

A set of extensions to the C language for dynamic and reflective control over function execution is described. A request function is a new kind of C function that allocates an explicit request context as part of its call. The request context includes all passed parameters along with additional data needed to interpret a request. Multiple functions may be dispatched against a single request context using an adapted form of "goto" statement. The request context can also be accessed with semantics identical to those of a memory object. These extensions are intended as a incremental addition to the underlying language, so that C can serve its common role as a "portable assembler." This role needs to include highly dynamic and reflective implementation of extensible services, such as those defined by an advanced object system.

1. Introduction

In a paper prepared for the 1990 ECOOP/OOPSLA reflection workshop [1], I proposed an initial version of C language extensions to support functions whose action could be assembled in highly dynamic ways under control of explicit reflective data. The extensions were successful in meeting their basic goals, and a variant of them (minus syntactic detail) remains in use by a C-based object system that adopts a generic function approach to defining object operations. They were also instructive in studying efficient implementation of fine-grained details of reflective control at the near-machine level of C. They did not succeed, however, at defining the underlying mechanisms in a simple and palatable enough form for wider consideration as a potential addition to the language.

This paper presents a simple and syntactically conservative set of extensions to the C language in hopes that they might be supported directly by existing C compilers. Direct implementation in a compiler is necessary to reduce the inherent penalties of reflective control to the minimum possible level. Machine-level implementation also remains interesting to reveal inherent costs and opportunities of reflective control. A key objective of these extensions is to ensure that no more penalty is paid for reflective control than actually needed or used. They are designed to enable many possible strategies and tradeoffs for managing reflective costs, such as the partial compilation of scripts noted in [2].

2. Objectives

The principal objective that continues to drive this work is the implementation, on a C language base, of object systems that adopt a generic function approach to supply all object behavior. Generic functions are the simplest possible method by which to add object-oriented programming capability to a host language (the function call mechanism already exists), and they preserve the simplicity of the base language for the application programmer. Additionally, techniques of dynamic method combination under control of explicit metaobjects, as in CLOS [3], can support an extreme degree of flexibility and variation in the implementation of functions defined by an object system.

Efficient but flexible mplementation of function-like requests is becoming more urgent with the emergence of distributed object management systems such as the OMG Object Request Broker [4]. Because these systems are accessible from multiple languages, they impose an essentially new object model into existing languages. The object model of the OMG CORBA architecture [4] is based on an especially strict separation of interface and implementation for object requests. An application for these C language extensions is to integrate as seamlessly as possible the heavyweight objects belonging to a distributed object manager with fine-grained objects belonging to a local object system.

A variety of other applications, including traditional roles for reflection like portable debugging facilities, were noted in the prior proposal [1]; the intended range of applications for this new proposal remains the same as before. This proposal, however, is more strict in reducing its scope to just a minimal set of mechanisms sufficient to increment C with a versatile form of dynamic control linked to extensible reflective data. A key goal has been for a simple but highly customizable extension that could be easily understood.

3. Proposed Extensions

The proposed extensions are based on a new class of function that allocates an explicit request context as part of its call. This request context contains the passed argument list and other optional data, and establishes a persistent stack context that may be accessed across multiple functions executing successively.

To distinguish them from conventional functions, functions declaring an explicit request context are referred to as “request functions,” and their declarations are distinguished syntactically by the presence of square brackets inside the parentheses surrounding a parameter list declaration. The parameter list declaration includes a full ANSI-style prototype, but may also declare additional variables to be allocated and retained as part of the request context. These additional members (if present) are separated from the main parameter list by a semicolon.

Calls to a request function look just like standard calls. A declaration for any called request function must be in scope at the point of a call, but the call itself contains no special syntax. Non-argument members of a request context are not passed or set by the call; all responsibility for their initialization and use belongs to the called function.

A simple and straightforward application of request functions is to select a single method to perform an object request. The method selection, for example, could be based on a dispatch table referenceable from within an explicit type object pointed to by each object instance. Figure 1 shows an example of such method selection; this example does not contain any non-argument members of the request context. Both the initial function and the selected method are declared as request functions of identical type.

```
typedef struct object_id *OBJECT, *STREAM; /* sample object types */

void printOn([ OBJECT anObject, STREAM aStream ]) /* initial function */
{
  goto anObject->type->methods[PRINT_METHOD_INDEX];
}                            /* select method using defined index*/


void samplePrintMethod([ OBJECT anObject, STREAM aStream ])
{
  addString(aStream, printString(anObject)); /* sample method action */
}
```

Figure 1. Dynamic selection of a single method using request functions.

In addition to the syntax for declaring request functions, Figure 1 illustrates the new control transfer capability provided for use inside request functions. A "goto" statement may be used to transfer control from one request function to another. Execution of the previously executing function is terminated, and all subsequent processing for the request is handled by the selected target function. The target function must have compatible type with the source.

The extended use of "goto" is much like the GNU C extension [5] that allows a goto target to be specified by a label variable, except that the target here has type "pointer to request function" and may lie entirely outside any file or lexical scope of the currently executing function. The selected function continues the computation in progress for the current request context, with no effect on the calling context where the current request will eventually return. This use of "goto" decouples the basic mechanism of control transfer from constraints of a conventional function call. It can also be thought of as applying or redispatching successive functions against a single context created by the initial call. The total action of a function can be constructed out of independent pieces dynamically selected and assembled end-to-end, with no changes to the overall stack context throughout the process.

The more complex example in Figure 2 makes use of non-argument members declared as part of the request context. This example implements a function "do_it" whose action is defined by before and after methods surrounding a primary method. All these methods are contained as pointers in an explicit object ("do_it_methods") that specifies the function.

```
typedef struct object_id *OBJECT;                /* sample object ID type */
typedef (*FPTR)([void]);          /* generic request function pointer type */
typedef (*MPTR)([ OBJECT, ...; FPTR, FPTR ]); /* do_it method pointer type */

extern void do_before([ OBJECT, ...; MPTR selectNext, MPTR *current ]);
extern void do_after ([ OBJECT, ...; MPTR selectNext, MPTR *current ]);

struct {
  MPTR *before_methods;         /* vector of request function pointers */
  MPTR primary_method;          /* single pointer to request function */
  MPTR *after_methods;          /* vector of request function pointers */
} do_it_methods;                /* contents initialized elsewhere */

void do_it([ OBJECT anObject, ...; MPTR selectNext, MPTR *current ])
{
  selectNext = (MPTR) do_before;
  current = do_it_methods.before_methods;
  goto do_before;
}

void do_before([ OBJECT anObject, ...; MPTR selectNext, MPTR *current ])
{
  if (*current != 0) goto *current++;

  selectNext = (MPTR) do_after;
  current = do_it_methods.after_methods;
  goto do_it_methods.primary;
}

void do_after([ OBJECT anObject, ...; MPTR selectNext, MPTR *current ])
{
  if (*current != 0) goto *current++;
  /* return to caller */
}
```

Figure 2. Method combination example.

The syntax for declaring pointers to request functions follows existing C syntax, with the addition of the square brackets. The do_it function as well as the do_before and do_after functions all specify a single initial argument plus a variable argument list portion, using syntax for function prototypes established by the ANSI C standard [6]. The additional members of the request context, following the semicolon, contain variables whose state is maintained across all goto control transfers between functions. The functions shown use these variables to maintain a current position in vectors of before or after methods.

The functions shown comprise an interpreter for a function whose specific action is determined by the contents of the data object do_it_methods. All methods called would need to declare their request contexts with compatible member types, and would also need to follow common conventions for their return action. Instead of executing an explicit return, they would need to execute the statement “goto selectNext” thus returning to the interpreter function that has control over subsequent action. Specification of common conventions across a family of interpreter and base methods is typically required as part of reflective function design. The choice of such conventions, however, is left open by the language extensions to enable a wide range of efficiency vs. flexibility tradeoffs.

Other important aspects of the extensions are not shown by these examples. A request context is a persistent object in its own right, and addresses of its components may be taken and passed as pointers for use in other functions as long the call which allocated the request context has not yet returned. Two special expression constants, “[]” and “[...]” are added to the language to refer to the fixed vs. variable portions of the request context for the currently executing request function. Figure 3 shows a sample use of these constants.

```
do_it_method([ OBJECT anObject, ...; MPTR selectNext, MPTR *current ]);
{
  print_do_it_request( &[], &[...] ); /* pass request context addresses */
  goto selectNext;
}

print_do_it_request( struct do_it_method *fixed, void *variable )
{
  /* print representation of do_it request using context pointers */
}
```

Figure 3. Sample use of request context addresses.

Request functions are an entirely distinct type added to the C language, so they can specify their own rules for memory layout of the members inside a request context without disrupting existing rules (or lack thereof) regarding argument lists of conventional C functions. To establish a simple and consistent layout rule, these extensions specify that both fixed and variable portions of the request context be laid out with memory alignment and offsets exactly like that of structure members. The fixed portion, in fact, defines a special structure tag with the same name as the function, as shown in the declaration of the argument “fixed” in Figure 3. Non-argument members precede argument members so that arguments of a function can be partially specified. (This permits generic interpreter functions that omit reference to all or part of their passed argument list.) The variable argument list portion, if present, must be accessed using offsets of argument types that are known to have been passed, but such access does not require the special macros of the ANSI C standard [6]. Both fixed and variable arguments may have their values modified during execution of any request function, and these modified values must be preserved across multiple “goto” transfers between functions.

3. Implementation Notes

The entire proposed language extension consists of the declaration syntax for request functions, “goto” control transfer between request functions, and the ability to declare and access components of a request context using the two special constants and consistent memory layout rules. These extensions are entirely upwardly compatible with the C language as defined by the ANSI C standard [6]. Because the new capabilities are defined only for a special class of C function type, which is distinguished syntactically everywhere declared, they supply a pure addition to the language which need not break any existing capability or implementation characteristics.

Request functions can be implemented almost exactly like conventional functions. Request contexts can be allocated on a stack just like conventional passed arguments and local variables. Register-based conventions for argument-passing, common on RISC architectures, can be used for request context members just as for conventional arguments. Register-based values need be flushed to a memory backup location only if their address is taken. The only slight performance impact compared to conventional register usage is to reserve reuse of all registers that could hold request context values, since these values must all be retained until a final return is about to be taken.

The goto control transfer can typically be implemented by a simple direct or indirect branch, perhaps coupled with a deallocation and reallocation of local variables. Constant address targets can be distinguished from the dynamic targets of true function pointers, and can be scoped in the same or different source file, so that optimal or necessary forms of branch linkage can be employed.

These levels of machine-level efficiency can be attained only by direct implementation within a compiler, but implementation is also possible using a preprocessor that generates standard C code as output. Under this class of implementation (the only one feasible for pure users of existing C compilers), request contexts are allocated as local structure objects, and addresses of these structure objects are passed to request functions instead of a conventional argument list. Argument values are then all accessed by indirection through the structure pointer rather than directly. An initial stub function intercepts the initial call, allocates the context object, and handles control flow transfer among subsequently executing functions. This form of implementation, though considerably less efficient than a native compiler implementation, has delivered adequate performance for a late binding object system used successfully to deliver several object-based applications of significant size.

References

[1] Burkhart, R., “Reflective Functions for the C Language”, in *Reflection and Metalevel Architectures in Object-Oriented Programming* (Informal Workshop Proceedings, Mamdouh Ibrahim, ed.), ECOOP/OOPSLA ‘90, Ottawa, Canada 1990

[2] Masuhara, H., et al, “Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently,” in *OOPSLA '92 Proceedings*, Association for Computing Machinery, 1992

[3] Kiczales, G., des Rivieres, J., Bobrow, D., *The Art of the Metaobject Protocol*, MIT Press, Cambridge, MA,1991

[4] *The Common Object Request Broker: Architecture and Specification (Revision 1.1)*, Object Management Group, Framingham, MA, 1992

[5] Stallman, R., *Using GNU CC (for version 2.0)*, Free Software Foundation, Cambridge, MA, 1992

[6] *American National Standard for Information Systems - Programming Language C*, Document X3.159-1989, ANSI, 1990