3-21-1986

Functions Returning Values of Dynamic Size

Mark Sherman
Dartmouth College

Andy Hisgen
DEC Systems Research Center

Jonathan Rosenberg
Carnegie Mellon University

David Alex Lamb
Queen's University - Kingston, Ontario

FUNCTIONS RETURNING VALUES OF DYNAMIC SIZE

Mark Sherman, Andy Hisgen,
Jonathan Rosenberg, David Lamb

Technical Report PCS-TR86-125

Functions Returning Values of Dynamic Size

Mark Sherman
Department of Mathematics and Computer Science
Dartmouth College
Hanover, NH. 03755

Andy Hisgen
DEC Systems Research Center
130 Lytton Ave.
Palo Alto, CA. 94301

Jonathan Rosenberg
Information Technology Center
Carnegie-Mellon University
Pittsburgh, PA. 15213

David Alex Lamb
Department of Computing and Information Science
Queen's University
Kingston, Ontario, Canada
K7L 3N6

February 1984, Revised March 1986

Abstract

Modern programming languages, such as Ada [Ichbiah 80], permit the definition of functions that return values whose size can not be determined until the function returns. This paper discusses five implementation techniques that can be used to implement this capability. Comparisons of the techniques are provided and guidelines for selecting a particular technique for a compiler are given.

Introduction

Modern programming languages, such as Ada, provide mechanisms for type extension, including ways to define values whose storage size is unknown at compile time. These values are usually called dynamic-sized values.

Although many languages provide dynamic-sized variables and dynamic-sized parameters, few allow dynamic-sized values to be returned as function results. Therefore, little effort has been expended to find and discuss efficient ways of implementing functions that return such values. For example, most popular compiler construction textbooks do not discuss this problem [Aho 73, Aho 86, Cheatam 67, Cocke 70, Gries 71, Lee 67, Lewis 76, Pollack 72, Pratt 75, Weingarten 73]. We were able to locate only one textbook that devotes even a paragraph to the problem [Hill 74].

Two languages that do allow dynamic-sized function-return values are Algol-68 [VanWijngaarden 69] and Ada. They illustrate two philosophies of run-time systems. The approach used by some (but not all) Algol-68 systems favors the heap for returning dynamic-sized function-return values [Hibbard 76,

Knueven 75]. Languages such as Ada are designed to be used with a classical stack run-time system. Because most hardware is stack rather than heap oriented, future languages and compilers will probably imitate Ada implementations. Therefore, techniques for returning dynamic-sized values in a stack environment are needed.

This paper provides an analysis of techniques for this problem. We first elaborate the problem and show how the stack model used for block structured languages is inadequate for returning dynamic-sized values. We then present five techniques that can be used for a run-time design to solve this problem. Finally, we comment on experiences with the various techniques.

The Problem

In a stack model, functions are a generalization of a stack operator: the operands (including the activation record for a called function) are pushed on the stack, then the operation is performed, popping the operands and pushing the result. Functions require this model to be extended; if a function were to place its result on the top of the stack before returning, the returned result would be popped from the stack along with the activation record of the function.

In practice, the returned result of the function must reside in a location that will remain after the function exits. One method for returning a function value reserves stack space for the function result before the activation record of the function is created [Moss 78]. This space resides between the activation record of the caller and the activation record of the called function. This space may also be a compiler-generated temporary in the activation record of the caller. The function places the result of its invocation into this reserved space. When the function has completed, its activation record is popped, leaving its return value on the top of the stack.

Another method uses a register to hold the value to be returned, with the caveat that the register will not be changed when returning from the function [Gries 71].

These methods rely on the ability to determine the size of the value to be returned at the time the function is called. The proper amount of storage must be reserved before the activation record for the function is created. In general, this value cannot be determined at compile time. As an example, consider the following Ada function:

```
type String is array(Natural range <>) of Character;

function Concat(X,Y: String) return String is
        S: String(1..X'Length + Y'Length);
begin
        S(1..X'Length) := X;
        S(X'Length+1..S'Length) := Y;
        return S;
end Concat;
```

Different calls of the function *Concat* may return strings of different sizes. The size of the returned string cannot be determined until the local variables of *Concat* are elaborated. Without a very intelligent compiler, the size of the returned value is not known until the actual return, and in general, there are programs for which no compiler can determine the size of the return value before executing the return. Therefore it is not possible either to reserve the necessary storage on the stack before the function is called, or to

guarantee that enough registers will be free for holding the returned value.

A Collection of Solutions

This section presents five solutions to the problem of returning dynamic-sized values as the results of functions. Each solution identifies a different place to keep the returned value. In all of the examples, we assume that *F*, *G* and *H* are functions that return dynamic-sized values.

Hole in the Stack

The first solution allocates the storage for the returned value on top of the activation record for the function. The stack before a function returns from a call is illustrated in Figure 1.

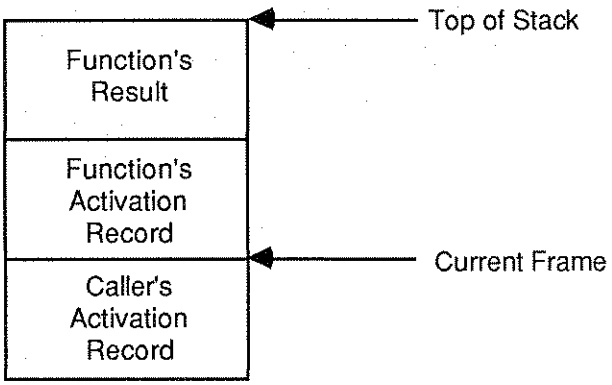


Figure 1: Before Function Returns with Value on Stack

When the function returns, the activation record for the function is *not* popped -- the stack pointer remains at the top of stack used during the execution of the function. Any frame pointer that indicates the base of the activation record for the current scope must still be updated. The stack after the function returns is illustrated in Figure 2.

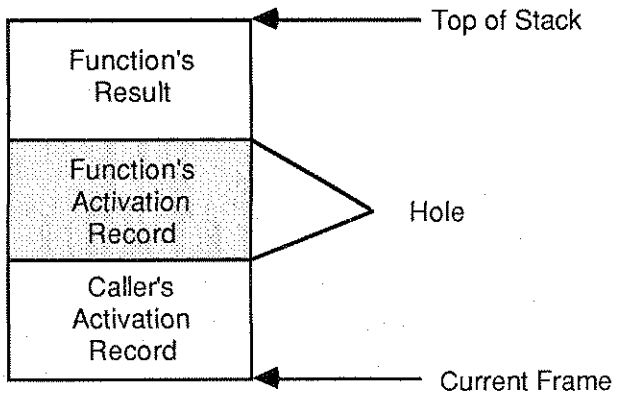


Figure 2: After Function Returns with Value on Stack

The returned value is preserved by not popping the stack at the end of the function call. Along with the returned value, the storage used by the function for its local variables remains allocated. However, this storage is inaccessible and constitutes the hole in the stack. The hole persists until after the returned value of the function is no longer needed. For example, in the assignment statement

x := F()

the return value must exist until the assignment has been completed. At that time, the activation record for the function and the returned value are both popped.

There are several disadvantages with this technique, however. First, it wastes stack storage. If large local variables are allocated by the called function, the space used by those variables will remain after the function returns. Deeply nested function calls can leave a large amount of wasted storage. Consider a function *F* whose return statement contains a call on another function, *G*. In an expression containing a call on *F*, the activation records for both *F* and *G* must be retained until the result of *F* is no longer needed.

Second, this scheme causes other stack oriented operations to become complicated. Consider a typical integer arithmetic expression, such as

j + k

In the stack model, the addition operation is accomplished as follows: push the left operand (*j*), push the right operand (*k*), pop the two operands and place the sum on the top of the stack. But if the expression uses a value returned with a hole in the stack, this is no longer possible. Consider the following arithmetic expression:

j + F(H(k), G())

where *G* returns an array and *F* and *H* return integers. Unless carefully implemented, referencing the top two integer locations on the stack will not yield the correct operands. As shown in Figure 3, a hole would separate them. (We assume that the integer-sized return value from *H* can be easily accomodated and is not shown.)

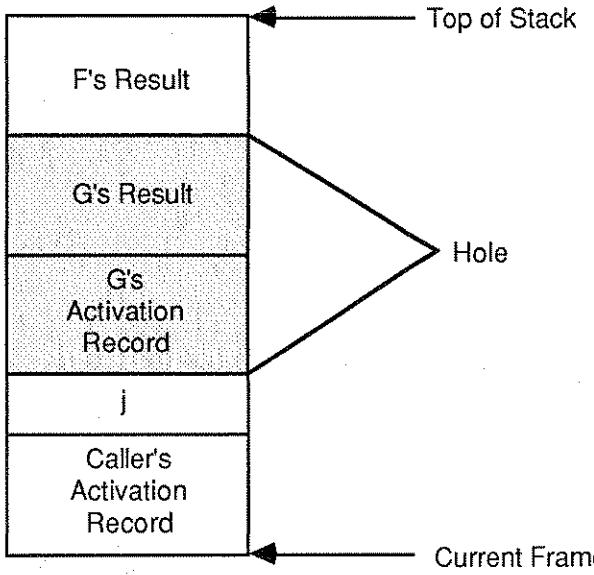


Figure 3: Interference of Stack Hole with Integer Addition

Thus even simple operations such as integer addition and assignment depend on the kind of functions used in subexpressions. For correct behavior, the stack space for *G*'s result must be deallocated at exactly the right point in expression evaluation: it must be retained during the call on *F*, and then deallocated right

before the *"+"*. The compiler must keep track of the context in which any particular dynamic-sized return value is used and pop the stack at the appropriate time. The expression above would be translated as follows:

```
push j;
save current stack pointer in old_SP;
push k;
call H;
pop result of H into temp1;
call G;
        -- G leaves the address of the array descriptor for its
        -- return value on the top of the stack.
pop address of array descriptor into temp2;
push temp1;
push temp2;
call F;
pop result of F into temp3;
set stack pointer to old_SP;
push temp3;
add;
```

The actual parameters to *F* needed to be evaluated and saved in temporaries, and then pushed on the stack just before the call. Because of the stack space consumed by *G*'s result, we could not leave the result of *H* on the stack to be referenced by *F* as a parameter.

Third, the scheme complicates the use of procedure-call hardware. Newer machines, such as the VAX [DEC 79], provide context switching instructions that automatically manipulate the stack pointer and other procedure context information. When a hole is left, the compiler must either perform all necessary context switching or undo some of the effects of the call instruction.

Sliding Value

The pure stack model requires that the returned value of a function reside on the top of the stack. This can be accomplished by the second method, which moves the returned value from where the called function left it to where the calling procedure expects it. This is similar to the hole-in-the-stack scheme except that after a function returns, the value that resides on top of the stack is moved to the top of the stack as it existed before the function call. The returned value is slid down the stack to the correct position. Figures 1 and 4 illustrate a stack before and after a function returns using this technique.

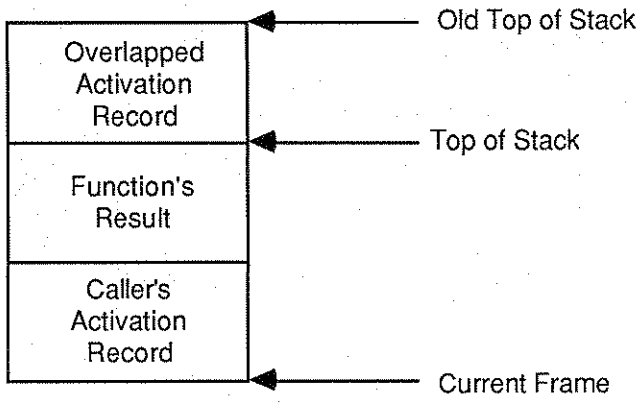


Figure 4: Sliding the Value Down the Stack

Sliding a value down the stack gives the program a consistent view of values returned from operations on the stack since the operands are always popped and replaced by the result. However, this solution has three drawbacks. First, this solution incurs execution overhead in sliding the value down the stack at each function return. Another problem with this method is possible interruption of the sliding operation. If the interrupt system uses the same stack as Ada programs, the stack may be in an inconsistent state when an interrupt procedure is called. The third problem is the incorrect encoding of data, such as internal pointers, in the returned value. The sliding of data down the stack resembles an assignment operation. Checks and translations must be made to ensure that any pointers in the returned value, or to the returned value, are preserved. Therefore the sliding operation may not be a simple block copy.

Stack Splitting

The previous methods use a single stack for all data, activation records, and intermediate expressions. However, other versions of the stack model permit different kinds of values to be placed on different stacks. One such model distinguishes between values whose sizes are known at compile time (static size) and values whose sizes are not known until run time (dynamic size). Descriptors, scalars, return addresses, static and dynamic links, pointers to parameter lists, some arrays and some structures are allocated on the static stack; dynamic arrays, records that contain dynamic arrays and union typed objects are allocated on the dynamic stack [Birrell 77].

Using this model, some of the pitfalls of the previous two methods can be avoided. The function invocations and exits, along with ordinary arithmetic, take place on the static stack, preserving the strict stack discipline for maintaining static links, displays, dynamic links and other attributes associated with block structured languages. This also permits the use of special hardware instructions that assist the implementation of function calls.

Dynamic-sized values are always placed on the second stack, regardless of whether these values are to be values which will be returned or local variables of the function being executed. This dynamic stack is cut back (the dynamic-sized values are popped) at a time when the dynamic values on the stack are unneeded. In most cases, this is the same time as when the static stack is popped. When a value is returned by a function, the dynamic stack remains unpopped until the returned value is used in the expression containing the function call. The stages of this method are illustrated in Figures 5, 6, and 7.

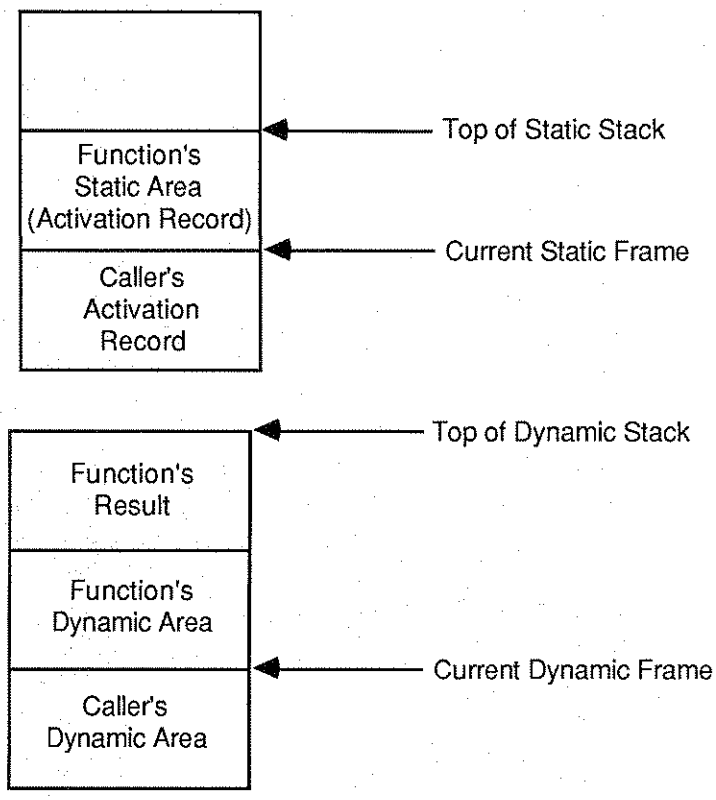
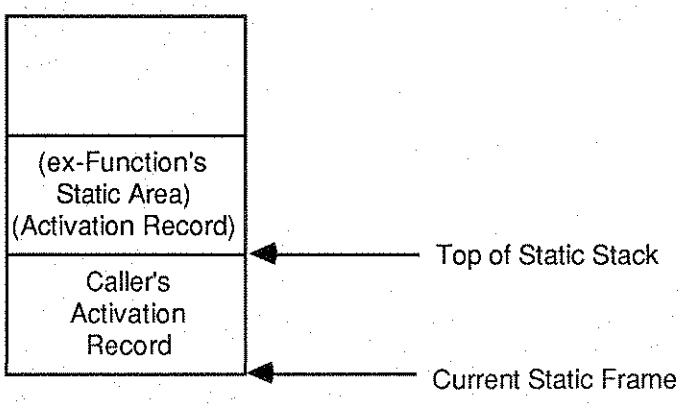


Figure 5: Split Stack -- Just Before Function Return



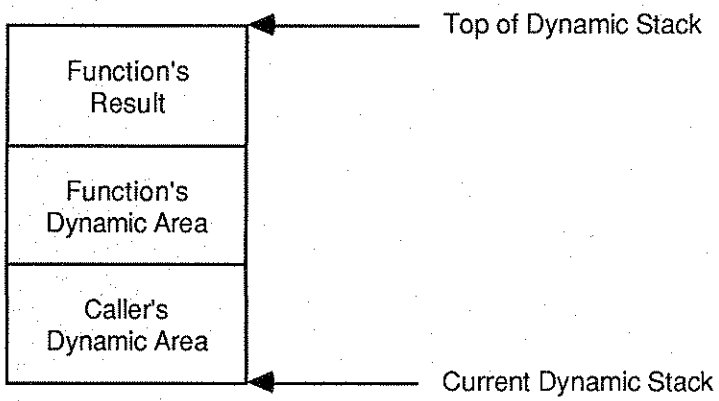


Figure 6: Split Stack -- Just After Function Return

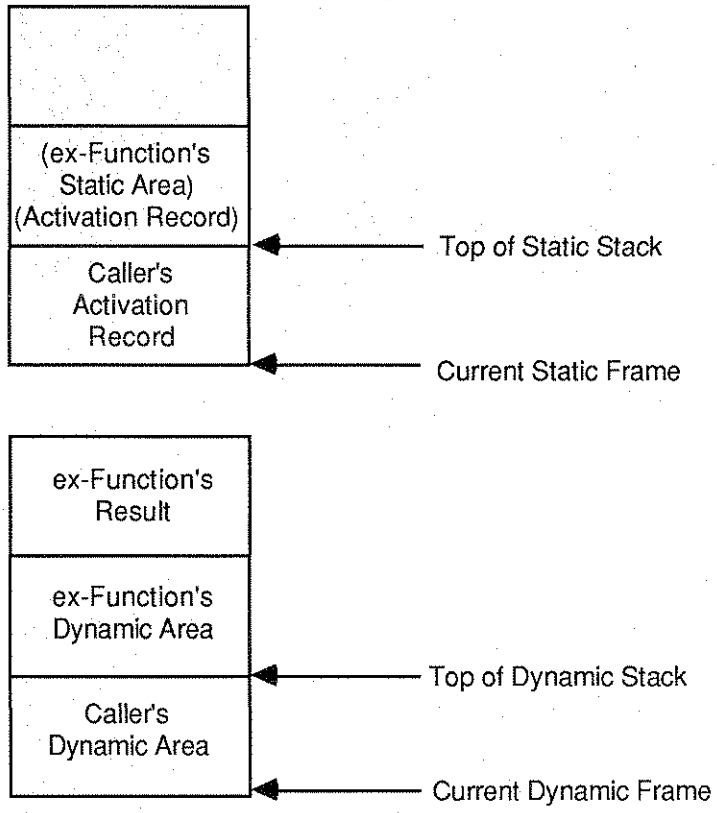


Figure 7: Split Stack -- After Return Value is Used

Although the static stack uses stack space efficiently and wastes no time copying values needlessly, holes can develop in the dynamic stack. Their space may be recovered when the space for the returned value of the function is recovered.

This method of allocating storage space obviously requires two stacks. It may be cumbersome on machines that cannot handle multiple stacks efficiently or in a run-time system that grows a heap and a stack towards each other.

Heap

The earliest solution to the problem of returning a function result that had a dynamic size probably used the heap as a place to keep the returned value. Because the heap is unaffected by stack operations, such as popping an activation record during the return of a function, it is a safe place for function results. A pointer to the value in the heap can be passed back to the calling routine in the same way as any other value with a predetermined size. Illustrations of the stack and heap before and after a function return are shown in Figures 8 and 9.

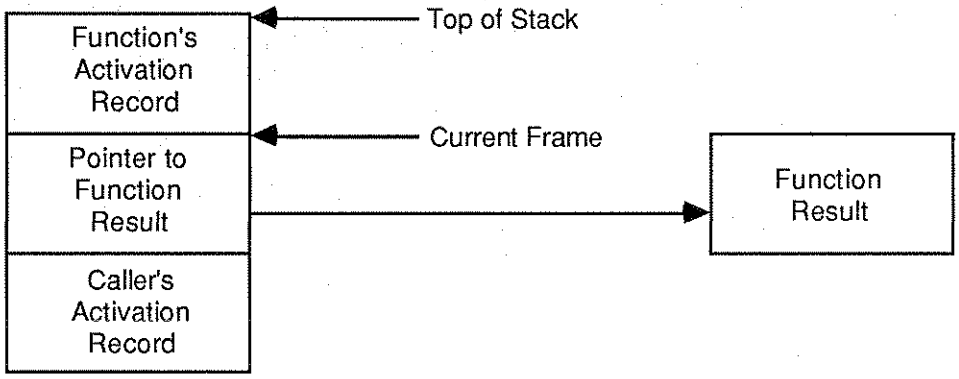


Figure 8: Before Function Returns using Heap

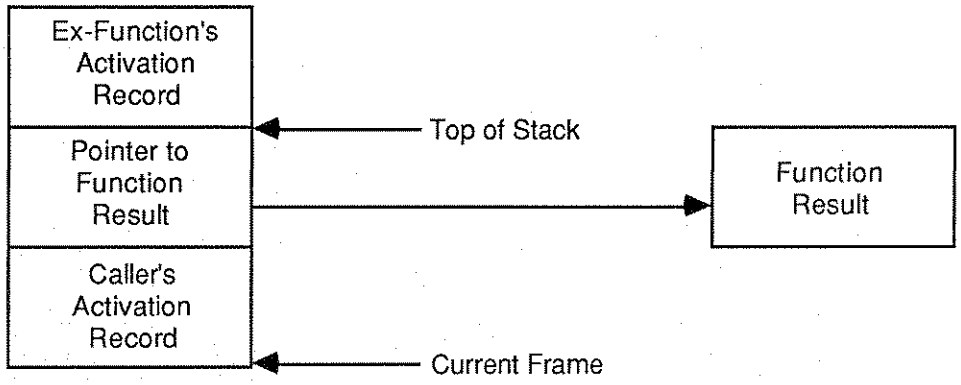


Figure 9: After Function Returns using Heap

Although this technique requires heap management, no garbage collection is required. The compiler creates all pointers to a function-return value and knows when the value is no longer needed or referenced. The compiler can explicitly deallocate the storage occupied by the return value.

An obvious implication of this method is the necessity of a heap. Although Ada and Algol-68 have heaps for other reasons, one can envision a language that has values returned by function that have dynamic sizes but has no user-controlled heap. Two such examples would be True Basic [Kemeny 85] and Algol-60 with functions that can return variable-length strings.

Using the heap can complicate the stack model just as the hole-in-the-stack can. The compiler must treat values of the same type differently because of their context. Consider the following example:

```
A,B: String(1..N);
X,Y: String(1..2*N);
...
X := Y;
X := Concat(A,B);
```

In the first assignment statement, the value of Y is available on the top of the stack for assignment to X. The second statement is also a simple string assignment, but the source string is in the heap, not the stack.

Stack Switching

Another two stack model uses the two stacks for a different set of purposes: one stack is for operands and the other is for results [Swierstra 79]. The operands are pushed on one stack, the operation is performed on the operands and the result pushed on the second stack, then the operands are popped from the first stack.

Rather than dedicating one stack for operands alone and the other for results, the model alternates the interpretation of the stacks. In particular, the interpretation of the stacks is switched when calling a nested function. This is illustrated by the following example. Consider the statement:

```
i := F(1,G(2,H(3),4),5);
```

To see the implicit nesting of function calls, we rewrite this statement using a prefix form:

```
:=(i,F(1,G(2,H(3),4),5);
```

We assume that the functions *F*, *G*, and *H* return arrays whose sizes are not known when they are called. We initially select stack *L* (for *left*) as the operand stack and stack *R* (for *right*) as the result stack. Both stacks are empty. The first procedure call is the assignment operator (:=). Its first operand, the address of *i*, is pushed on the stack. This is illustrated in Figure 10.

Figure 10: Stack Switching -- After First Operand is Stacked

Next, a nested call to *F* is made, so the stacks must be switched. Stack *R* now becomes the operand stack and stack *L* becomes the result stack. The first parameter of *F*, the value *1*, is pushed onto stack *R*. This is

illustrated in Figure 11.

Figure 11: Stack Switching -- First Operand of Nested Call

Another nested call is made, so the stacks are switched again. Stack *L* becomes the operand stack and stack *R* becomes the operator stack. The first parameter of *G*, *2*, is now pushed on stack *L*. The resulting stacks are shown in Figure 12.

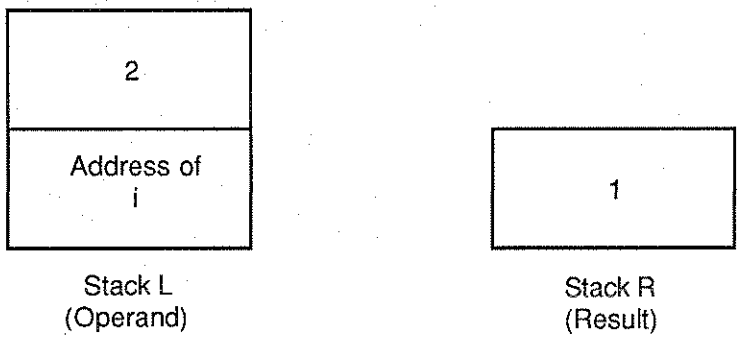


Figure 12: Stack Switching -- Several Pending Function Calls

The last nested function is found, so again the stacks are switched. The parameter *3* is pushed on stack *R* and function *H* is called. Because the activation record for *H* can be thought of as another parameter to *H*, it is allocated on the operand stack, that is, stack *R*. (Note: procedure calls within *H* are "nested" in *H*, so the stacks would be switched again.) The stacks just before *H* starts executing are shown in Figure 13.

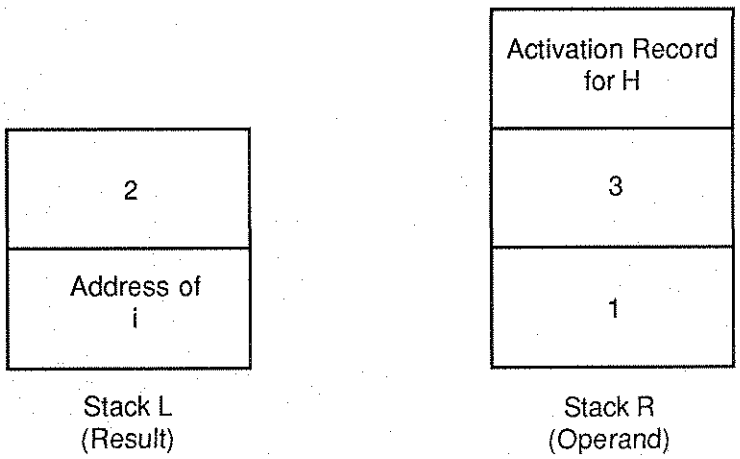


Figure 13: Stack Switching -- Just Before Function H Starts Executing

When *H* returns, it places its result on its result stack, stack *L*. The function return pops off the activation record and parameters for *H* from the operand stack. The stack after *H* returns is shown in Figure 14.

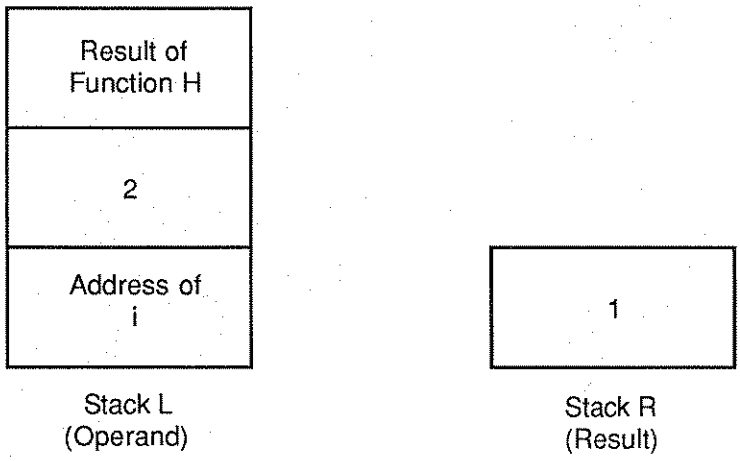


Figure 14: Stack Switching After Function H Returns

The result from *H* is the second parameter for *G* and it resides just above the first parameter for *G*. For the function call of *G*, stack *L* is the operand stack, so the third parameter for *G*, *4*, is pushed on stack *L* and *G* is called. The results of *G* are returned on stack *R*. The state of the stacks after *G* returns is shown in Figure 15.

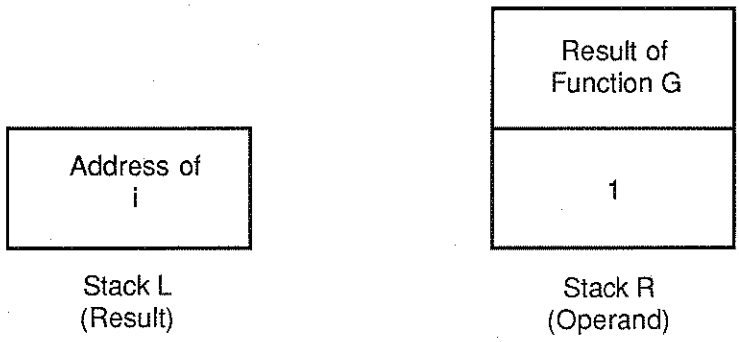


Figure 15: Stack Switching -- Switching the Result Stack

The result of *G* is the second parameter to *F*. Next the third parameter is pushed on the operand stack for *F*, that is, stack *R*. The result of *F* is left on stack *L*. The state after *F* returns is shown in Figure 16.

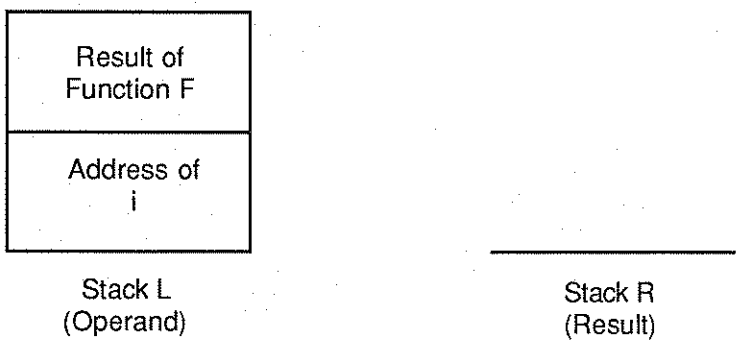


Figure 16: Stack Switching -- Just Before Assignment

The last procedure to be executed is the assignment operation. Note that the two operands for assignment are stacked correctly on stack *L*: the address of the variable and the value to be assigned. When the assignment is completed, the stack is popped and two empty stacks are left.

Note throughout this algorithm that no explicit copying or sliding of values is done. No holes are left in any stack. All deallocation is performed implicitly and correctly when a function exits.

Although superficially superior to the other methods, this technique has a couple of drawbacks. One problem is the inability of most hardware to support multiple stacks efficiently. Many implementations allocate one end of memory for the stack and have the heap grow towards the stack. This technique leaves no reasonable place to put the second stack.

A hidden problem is the location of intermediate results. Results are not always on the stack where they will ultimately be returned. They must be moved from one stack to another. This causes a great deal of copying when each function merely returns the result of an inner function call, for example,

```
function F(I: Integer) return Integer is
begin
        ...
        return G(I);
end F;

...

J := F(3);
```

In the hole-in-the-stack scheme, an implementation might leave the returned result on the top of the stack from the innermost function call and not copy it every time a procedure returns.

Experience

Few languages currently have a need for implementations that can return values with dynamic sizes. So experience with the language feature and its implementations is rather limited. A brief review of existing implementations is given below.

There are Algol-68 systems that use the hole in the stack scheme [Branquart 76, Hibbard 74] and the heap

scheme [Branquart 70, Hibbard 76]. Other Algol-68 designs uses the sliding scheme, though the details of these designs are not provided in their descriptions [Hill 74, Meertens 76]. Another Algol-68 system uses dynamic and static stacks for holding different values, returning the results of functions on the dynamic stack [Birrell 77].

Both the Sail compiler from Stanford [Reiser 76] and True Basic [Kemeny 85] use the heap for their strings, which are the only values with a dynamic size that may be returned by functions in both languages. Because all strings are in the heap, this choice maintains a consistent run-time representation of strings. Icon [Griswold 83] uses the same philosophy: all dynamic-sized objects (in Icon, this means lists, tables, strings and sets) are allocated in the heap, and all dynamical-sized returned values are also in the heap.

The SIMPL family of languages also allowed strings to have lengths determined at run time and uses a hole-in-the-stack technique to return them from functions [Gannon 79].

The NYU Ada system uses a heap to hold all values [Dewar 80], including values returned by functions. There was no need therefore to consider stack alternatives for this specific problem. Two other Ada systems use the hole-in-the-stack method [Lamb 80, Wilcox 81].

A Pascal system was designed using the two stack model of computation [Swierstra 79]. This design investigated multistack models, and the presentation of stack switching is based on this work. A similar technique was proposed for an implementation of ASBAL [Moss 78], a stack-based data abstraction language related to CLU [Liskov 81].

Although all the techniques have been implemented, there seems to be little written about the usefulness of any particular technique. Birrell [Birrell 77] asserts that the holes in the hole-in-the-stack method can be vast and the copying in the sliding-value method can be very expensive but provides no empirical data. Such conclusions can be made only after a study of how a particular language is used. Unfortunately for our purposes of analysis, there is a synergistic effect between the use of a language feature and the perceived efficiency of its implementation. Features that are thought to be expensive are avoided. Unless an implementation for dynamically-sized return values is perceived to be efficeint, programmers will avoid the feature and we will not learn if the feature is useful enough to invest the effort to streamline its implementation. However, if values returned by functions nearly always have static sizes, then it matters little which technique is implemented. The evaluation we provide below is based on our experience in designing and writing an Ada compiler and run-time system [Lamb 80, Rosenberg 80].

In this paper we have adhered to a stack model for function calls. A compiler writer, however, will typically use special-case (ad hoc) techniques for the translation of certain built-in functions. For example, when implementing integer addition, an implementor will choose to use the machine registers and non-stack-oriented machine instructions. Nonetheless, for user-defined functions, the generality of the stack model is still needed to handle parameter passing, local storage allocation in the called function, recursion, and dynamic-sized return values.

We found that the complexity of the compiler does not vary much for each scheme. It requires the same design effort on the part of the compiler builder to keep track of when to deallocate a value from the heap, of when to cut back the stack to release the hole, of when and where to slide a value down the stack, and of how multiple stacks are implemented.

We did find, however, that small changes in the architecture strongly affect the choice. Machines without strict calling conventions can more conveniently handle the stack violations perpetrated by the

hole-in-the-stack scheme. The availability of multiple stack and frame pointers assist in using a multiple stack scheme. Sharing one stack for both interrupts and application programs complicates any scheme that requires some atomic operation that is coded with more than one machine instruction, for example, stack switching or value sliding. Such architectural considerations seem to drive the design more than any perceived language use by programmers.

Conclusions

We have shown that reasonable implementations are available for functions that return values with dynamic sizes. Each of these implementations has been used in a running system. The difference between the implementations is the location of the return value. Each of the methods discussed has about the same complexity for the compiler writer, so the decision to favor a particular scheme depends on the specific architecture of the target machine.

Acknowledgements

This work has been growing incrementally since the authors started working on Algol-68 and Ada. Many individuals have contributed bits and pieces that we can no longer identify individually. However, we would like to thank Peter Hibbard especially for his comments on various Algol-68 systems, Paul Hilfinger and Paul Knueven for their review of the Ada systems, and members of the Ada Implementors' Group, now called AdaTEC, for their review of earlier presentations of this work. Finally, we thank Cynthia Hibbard for providing a careful reading of the manuscript.

This research was started while the authors were at the Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA. 15213, and was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under contracts F33615-81-K-1539 and F33615-78-C-1551. Jonathan Rosenberg was supported in part by a Fannie and John Hertz Foundation fellowship.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the US Government, The Hertz Foundation, Carnegie-Mellon University, Dartmouth College, Digital Equipment Corporation or Queen's University. Actually, we're not sure we believe them ourselves.

Bibliography

[Aho 73] Alfred V. Aho and Jeffrey D. Ullman, *The Theory of Parsing, Translation, and Compiling*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1973, Two volumes.

[Aho 86] Alfred V. Aho,Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Inc., Reading, Massachusetts, 1986.

[Birrell 77] A. D. Birrell, "Storage Management for ALGOL68," In *Proceedings of the Strathclyde Algol 68 Conference*, pages 82-94. March, 1977. Also *SIGPLAN Notices*, Vol. 12, No. 6, June 1977.

[Branquart 70] P. Branquart and J. Lewi, "A scheme of storage allocation and garbage collection for ALGOL 68," In J. E. L. Peck (editor), *ALGOL 68 Implementation: Proceedings of the*

IFIP Working Conference on ALGOL 68 Implementation, pages 199-238. International Federation for Information Processing, Technical Committee 2, July, 1970.

[Branquart 76] P. Branquart, J.-P. Cardinael, J. Lewi, J.-P. Delescaille, M. VanBegin, *An Optimized Translation Process and its Application to ALGOL 68*, Springer-Verlag, New York, New York, 1976. Also Report R204, MBLE, Brussels.

[Cheatam 67] T. E. Cheatam Jr., *The Theory and Construction of Compilers*, Massachusetts Computer Associates, Wakefield, Massachusetts,1967.

[Cocke 70] John Cocke and J. T. Schwartz, *Programming Languages and Their Compilers*, New York University, New York, New York, 1970.

[DEC 79] Digital Equipment Corporation, *VAX-11 Architecture Handbook*, Digital Equipment Corporation, Maynard, Massachusetts, 1979.

[Dewar 80] Robert B. K. Dewar, Gerald A. Fisher Jr., Edmond Schonberg, Robert Froehlich, Stephen Bryant, Clinton F. Goss and Michael Burke, "The NYU Ada Translator and Interpreter," In *Proceedings of the ACM-Sigplan Symposium on the Ada Programming Language*, pages 194-201. ACM-Sigplan, Boston, December, 1980. Also *Sigplan Notices*, Vol. 15, Nol. 11, November 1980.

[Gannon 79] J. D. Gannon and J. Rosenberg, "Implementing Data Abstraction Features in a Stack-based Language," *Software Practice and Experience* , 9:547-560, 1979.

[Gries 71] David Gries, *Compiler Construction for Digital Computers*, John Wiley & Sons, Inc., New York, New York, 1971.

[Griswold 83] Ralph E. Griswold and Madge T. Griswold, *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ., 1983.

[Hibbard 74] P. G. Hibbard, "The Run-time Organisation of an Algol 68 Sublanguage Compiler," In P. R. King (editor), *Proceedings of an International Conference on Algol 68 Implementation*, Utilitas Mathematica Publishing, Winnipeg, Canada, 1974.

[Hibbard 76] P.G. Hibbard, P. Knueven, and B. W. Leverett, "A Stackless Run-Time Implementation Scheme," In *Proceedings of the Fourth International Conference on the Design and Implementation of Algorithmic Languages*, pages 176-192, June, 1976, Courant Institue of Mathematical Sceinces, New York.

[Hill 74] Ursula Hill, "Special Run-Time Organization Techniques for Algol 68," In Goos, G. and Hartmanis, J. (editor), *Compiler Construction: An Advanced Course*, chapter 3, pages 222-252, Springer-Verlag, New York, New York, 1974, Vol. 21.

[Ichbiah 80] Jean Ichbiah, et al., *Reference Manual for the Ada Programming Language,* US Government, Washington, D.C., 1980.

[Kemeny 85] John G. Kemeny, Thomas E. Kurtz and Brig Elliott, *True Basic: The Structured Language System for the Future*, Addison-Wesley Publishing Company, Reading, MA.,

1985.

[Knueven 75] Paul Knueven, "The Foundation of a Flexible Run-time System for ALGOL 68S," In C. C. Charlton and P. H. Lang (editor), *Experience with ALGOL 68, Proceedings of the Liverpool University Conference*, April, 1975.

[Lamb 80] David Alex Lamb, Andy Hisgen, Jonathan Rosenberg, Mark Sherman and Martha Borkan, *The Charrette Ada Compiler*, Technical Report CMU-CS-80-148, Carnegie-Mellon University, Department of Computer Science, October, 1980.

[Lee 67] John A. N. Lee, *The Anatomy of a Compiler*, Van Nostrand Reinhold Company, New York, 1967.

[Lewis 76] P. M. Lewis II, D. J. Rosenkrantz and R. E. Stearns, *Compiler Design Theory*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1976.

[Liskov 81] B. Liskov, E. Moss, C. Schaffert, R. Scheifler and A. Snyder, *The CLU Reference Manual*, Springer-Verlag, New York, N.Y., 1981, Lecture Notes in Computer Science No. 114.

[Meertens 76] L. G. L. T. Meertens, *A Space-Saving Technique for Assigning ALGOL 68 Multiple Values*, PhD thesis, Mathematisch Centrum, Amsterdam, 1976.

[Moss 78] John Eliot Blakeslee Moss, *Abstract Data Types in Stack Based Languages*, Technical Report MIT/LCS/TR-190, M.I.T., Laboratory for Computer Science, February, 1978.

[Pollack 72] Bary W. Pollack, *Compiler Techniques*, Auerbach Publishers, Princeton, New Jersey, 1972.

[Pratt 75] Terrence W. Pratt, *Programming Languages: Design and Implementation*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1975.

[Reiser 76] John F. Reiser, *SAIL*, Technical Report Memo AIM-289, Stanford Artificial Intelligence Laboratory, August, 1976, Also Computer Science Department report STAN-CS-76-574.

[Rosenberg 80] Jonathan Rosenberg, David Alex Lamb, Andy Hisgen and Mark Sherman, "The Charrette Ada Compiler," In *Proceedings of the ACM-Sigplan Symposium on the Ada Programming Language*, pages 72-81. ACM-Sigplan, Boston, December, 1980. Also *Sigplan Notices*, Vol. 15, Nol. 11, November 1980.

[Swierstra 79] S. D. Swierstra, *Machine Architectures for Block-Structured Languages*, Technical Report 262, Twente University of Technology, Department of Applied Mathematics, P.O. Box 217, 7500 AE Enschede, The Netherlands, May, 1979.

[VanWijngaarden 69] A. Van Wijngaarden, B. Mailloux, J. Peck and C. Koster, "Report on the Algorithmic Language Algol 68," *Numerische Mathematik* 14(2):79-218, 1969.

[Weingarten 73] Frederick W. Weingarten, *Translation of Computer Languages*, Holden-Day Inc., San

Francisco, California, 1973.

[Wilcox 81] Bruce Wilcox, "Ada/TOPS20 Compiler Project," *AdaLetters,* I-1.9, July-August, 1981.