Why is a goto like a dynamic vector in the BCPL-Slim computing system?

by

Harvey Abramson

Technical Report 80-9

November 1980

Department of Computer Science
The University of British Columbia
Vancouver, British Columbia V6T 1W5

Abstract

The Slim computer is a new virtual machine which can be used in the translation and porting of the BCPL compiler, and eventually, in the porting of an operating system written in BCPL. For the purposes of this paper, the Slim computer is a stack machine with a single accumulator and a register which points to the top of the stack. The procedures LEVEL and LONGJUMP, traditionally used to implement transfers of control across BCPL procedures, and which are usually written in the assembler language of a host machine, cannot be used with this architecture. In developing procedures to implement all transfers of control, we show how these essential procedures - though highly dependant on the Slim architecture - can be written portably in BCPL, and discover an interesting connection between implementing jumps and dynamic vectors (by means of Aptovec) in the BCPL-Slim computing system. Some parameters of portability in mapping an abstract machine to host machines are identified, and it is shown how to maintain the portability of the above mentioned procedures in the face of various mapping problems. Finally, we are led to a comment on the design of BCPL to the effect that goto's are an unnecessary feature of the language.

Research supported by the National Sciences and Engineering Research Council of Canada.

The Problem

The syntax and semantics of BCPL permit sensible unconditional transfers of control (goto's) from one point to another only within the same procedure (i.e., routine or function). Each procedure call results in a new dynamic environment or stackframe which contains a linkage area to the previous environment, arguments to the procedure, local variables and vectors. A BCPL label is a static (or global) variable initialised to a value which represents the point of the program at which it is declared. If a procedure F has a label L, then goto L is a sensible command within F which results in a transfer of control to the point designated by L. Within another procedure G, however, the command goto L would result in a transfer of control to the point labeled by L, but within the environment of G. References to parameters and local variables meant to be in F would actually refer to cells in the environment of G, leading quite naturally to disaster.

In cases when an unconditional transfer of control was required between points in different procedures F and G, BCPL historically used a pair of procedures named LEVEL and LONGJUMP to effect the change: a call in F to the function LEVEL() is used to record a BCPL value representing F's environment in a static or global variable, say SaveE;

```
SaveE := LEVEL()
```

and then a call in G of the routine LONGJUMP causes the BCPL runtime system to restore the saved environment and to resume flow of control from the point labeled by L in F

```
LONGJUMP(SaveE,L).
```

Examples of the use of LEVEL and LONGJUMP may be found by examining the source programs of the BCPL compiler and debug package, and are discussed in [Richards & Whitby-Strevens, 1979].

The above scheme for unconditional transfers across procedures is not useable without modification on the Slim machine, which is a new abstract computer intended to serve as a target for the BCPL compiler and as an aid to the porting of the BCPL compiler and also, eventually, of a single-user multi-process operating system. (See [Abramson et al, in preparation].) The Slim computer is a stack machine with a single accumulator (the A register), and with a register (the H or high-point register) which points to the top of the stack. This architecture was chosen because code generated for the stack-accumulator machine is generally more compact than code for a pure stack machine; and because the presence of the H register makes it possible to treat interrupts as unexpected

procedure calls and allows the easy implementation of dynamic vectors in addition to facilitating the storage and retrieval of implicit variables during expression evaluation. (See [Fox, 1978].) The LEVEL and LONGJUMP pair of procedures cannot be used without alteration because the Slim computer requires not only that the environment be re-established, but also that the H register point to the correct cell on the stack when flow of control resumes at the destination label. The solution to this problem points out an interesting connection between unconditional transfers and dynamic vectors on the Slim computer, makes possible a more portable BCPL library, and suggests some comments about BCPL and about portability via the abstract machine technique.

A Solution

In preparation for an unconditional transfer of control on the Slim machine, the environment is recorded not in a simple BCPL value, but in a vector of three cells by a BCPL function called Remember.

In order to understand the Remember function, however, it is necessary to describe the Slim computer's representation of a procedure's environment. The current environment or stackframe is pointed to by the environment or E register. The E register in fact points to the second cell of the two cell linkage area of the current stackframe, in which is recorded a return address or value for the Slim computer's program counter or C register (which is where the calling procedure resumes upon normal termination of the called procedure). In the first cell of the linkage area, at an offset of -1 from the E register is a value which points to the stackframe of the calling procedure, that is, it contains the value of the E register at the calling point. If the current procedure has n arguments, these are found on the stack at positive offsets 1,...,n from the E register. Any local or vector variables lie on the stack above the argument(s), and above these lie the elements of the respective vectors.

Using the following manifest constants

```
manifest { LinkSize  = 2
           ECell     = 0
           CCell     = 1
           SaveE     = 0
           SaveLabel = 1
           SaveH     = 2
         }
```

the BCPL recording function Remember is given by:

```
let Remember(Recover, Label) = valof
{ Recover!SaveE := (@Recover-LinkSize)!ECell || save old E
  Recover!SaveLabel := Label                 || Transfer target
  Recover!SaveH := @Recover - LinkSize - 1   || save old H
  resultis Recover
}
```

The arguments to the function Remember are Recover, the three cell vector in which the information necessary for transfer of control is to be recorded, and Label, the target of the eventual jump. Within the funtion Remember, the absolute address on the stack of Remember's linkage area is given in BCPL by:

@Recover - LinkSize.

The value of the E register when Remember is called is therefore given by:

(@Recover-LinkSize)!ECell,

and the value cf the H register when Remember is called is given by:

@Recover - LinkSize - 1,

which is the address of the cell just below the current environment's linkage area. The result of Remember, the address of the recording vector, should be stored in a static or global variable.

The Transfer routine which actually performs unconditional jumps depends on the way that the BCPL return construct is implemented on the Slim computer. On Slim, a return from a procedure has the following effects: the E and C registers are reset to the respective values found in the current stackframe's linkage area; and the H register is reset to point to the cell just before the current stackframe's linkage area. There is no difference in a return from a routine or a function. The result of a function is simply found in the A register or accumulator.

Using the following static variables

```
static { NewLink     = 0
         RecoverCopy = 0
       }
```

the BCPL Transfer routine is given by:

```
let Transfer(Recover) be
{ let OldLink = @Recover - LinkSize
                                    || New link just above old H.
  NewLink := Recover!SaveH + 1      || Use static variables
  RecoverCopy := Recover            ||  to prevent overwriting.
  if OldLink = NewLink              || A goto in same environment?
  do { NewLink!CCell := Recover!SaveLabel
       return
     }
  || Set ECell to old H + LinkSize,
  || force return within Transfer.
  OldLink!ECell := Recover!SaveH + LinkSize
  OldLink!CCell := ReturnHere
  return
ReturnHere:
  NewLink!ECell := RecoverCopy!SaveE      || set ECell to old E
  NewLink!CCell := RecoverCopy!SaveLabel  || set CCell to target
}
```

Transfer works by manipulating its own linkage area and by setting up a "virtual" linkage area from which a return can be made to the appropriate label, in the appropriate environment, and with the appropriate value in the H register. The address of Transfer's linkage area is recorded in OldLink, and the virtual link, recorded in NewLink, is to be created one cell beyond the value of the H register saved in the Recover vector by Remember. The address of the Recover vector is saved in the static variable RecoverCopy (see below). Transfer's linkage area is first adjusted so that at the explicit return, instead of returning to Transfer's calling point, control resumes at the command within Transfer labeled "ReturnHere", and with the E register pointing to the virtual NewLink. The virtual linkage area is then set so that it contains the appropriate (Remember-ed) value for the E register, and the Label to which control should be transferred. The implicit return at the end of Transfer uses this virtual link to set the E, C, and H registers appropriately. In the case where there is no change of environment and the transfer is equivalent to a goto, i.e., when OldLink and NewLink have the same value, it is only necessary to reset the return address in Transfer's linkage area.

Needless to say, this manipulation of the stack is not without cost and risk. The variables NewLink and RecoverCopy must be static rather than local because at the return within Transfer the environment is damaged and local variables in Transfer could not properly be accessed by offsets from the E register. Furthermore, when the OldLink and NewLink areas overlap but are not coincident, there is a temporary stack anomaly upon the return within Transfer in that the H register is pointing to a cell below the cell pointed to by the E register. There is a stack protection mechanism on the Slim

computer, however, which readjusts the H register to the same value as the E register so that the condition which should hold between these registers

$$H \geq E$$

is in fact satisfied.

A Bonus: APTOVEC in BCPL

In BCPL the size of a vector must be known at compile time, that is, statically. The fixed size of the vector is used by the compiler to determine the exact size of the environment in which the vector is declared. If a vector v is declared in a procedure F then a cell is allocated on the stack to the vector variable v. This cell lies beyond F's arguments and among F's local variables, and is initialized to point to the first cell (v!0) of an array of n+1 cells. This array of cells lies beyond all of F's local and vector variables. Because of the fixed size requirement, programmers are forced to define vectors of such a size as to cover all expected eventualities. In those cases where programmers required or wished to specify vector sizes by some (dynamic) computation, a function APTOVEC was provided which could be described, albeit illegally, by the following BCPL-like code:

```
let APTOVEC(F,N) be
{ let V = VEC N || illegal!
  resultis F(V,N)
}
```

According to [Richards & Whitby-Strevens, 1979], this function is "normally implemented in assembly language" simply because dynamically determined vector sizes as given in the pseudo-definition of APTOVEC are not permitted in BCPL. Space for the vector would be dynamically allocated within APTOVEC's stackframe, just below the environment for F. Upon an exit from F, the space would be automatically deallocated when APTOVEC's stackframe was destroyed.

It is possible to define a version of APTOVEC for the Slim computer in BCPL in the absence of dynamically (i.e., run time) determined vector sizes by making use of the Remember and Transfer procedures. The "trick" is to suitably manipulate a Remember-ed value of the H register, and to force a Transfer within the following function:

```
let Aptovec(f, n) = valof
{ let Save = vec 2
  and v    = vec 0
  Remember(Save, LabelA)
  Save!SaveH +:= n    || This adds n to the saved H value
  Transfer(Save)      || producing the effect of v = vec n.
LabelA:
  resultis f(v, n)
}
```

Just before the call to Remember the H register is pointing to the single, zero'th element of the vector v. It is this value of the H register which is Remember-ed. Immediately after the call to Remember, the saved value of the H register is incremented by n (the notation x +:= y being syntactic saccharine for x := x + y). The call of Transfer is like a goto the following command, but has the side effect of altering the H register and therefore increasing the size of the vector v to n + 1 elements. Upon a normal exit from f, the implicit return in Aptovec restores the environment of the procedure which called Aptovec, and thereby, deallocates the space given to the vector v and resets the H register to its value at the calling point. (See below, however, for another more efficient version of Aptovec which, although it makes use of the Slim computer's assembly language, is still portable.)

The answer to the question which titles this paper is therefore: A dynamic vector is like a goto in the BCPL-Slim computing system because both dynamic vectors and goto's can be implemented by means of a common data structure and a common set of procedures which manipulate this data structure.

Some Parameters of Portability

Historically, BCPL compilers have been ported by the abstract machine technique, whereby code is generated for a machine which is interpretively simulated on a host computer. The first such BCPL machine was the OCODE machine ([Richards, 1971]), succeeded by the INTCODE machine ([Richards, 1974]). More recently the Pica-B ([Abramson et al, 1978]) and the Slim ([Abramson et al, in preparation]) computers have been designed to aid in the porting of the BCPL compiler and also, eventually, in the porting of operating systems. (The Pica-B is essentially the INTCODE machine with an interrupt register.)

The INTCODE machine was designed to simplify the portability process because when programmers with no previous experience of BCPL undertook to port the compiler by means of

the OCODE machine, it was found that "it often took longer than expected and frequently ... strategic errors in design" were made ([Richards, 1974]). Even though the INTCODE machine may have eased the problem of porting the BCPL compiler, there still seems, however, to have been an important gap in the portability process, namely the existence of important library procedures not written in BCPL: typically the procedures LEVEL, LONGJUMP and APTOVEC were written in the assembler language of the computer hosting the BCPL compiler. The successful implementation of these procedures without an explicit operational defintion of their effects could prove an obstacle (albeit a slight one to experienced programmers) to the porting of BCPL from one machine to another.

Given an abstract machine such as Slim, for example, it is relatively easy to write in BCPL important library procedures which, though highly machine dependent, become as portable as the BCPL compiler itself. This is not to say that eventually, after a successful port of the compiler and library, more efficient versions of these procedures should not be written in the assembler language of the host machine. Rather, the initial port becomes much simpler in that one is provided with procedures which provide high-level, reliable and explicit operational descriptions of sometimes non-standard or even bizarre manipulations of the abstract machine.

There is another advantage to portable versions of such procedures and that is that they provide a test of the consistency of the mapping of the abstract machine to the host machine. If the mapping of the abstract machine is straightforward and correctly done, then these procedures - unaltered - should behave as expected. Inconsistencies in the behavior of these procedures on the host machine would be an indication of a possible error in the mapping.

The host machine, however, usually has one or more architectural quirks which must be accounted for in any mapping of the Slim machine, and some compiler implementation decisions may, in effect, be viewed as perturbations of the Slim computer. It is possible to revise the definitions of the procedures given above in such a way that they perform their expected functions as defined on the Slim computer yet are still portable even in the face of machine specific or implementation specific problems.

Several parameters of the mapping of Slim to a host machine can be identified:

1. The linkage structure for procedures may be different: there may be more than two cells in the linkage area and/or the arrangement of cells in the linkage area may be a permutation of the order given above.

2. There may be a hardware stack or stack register which imposes a "downward" growing stack: when something gets pushed onto the stack the value in the H register decreases; and when something gets popped, the value in the H register increases.

3. The addressing mechanism on the host machine may be different from the BCPL-Slim cell addressing scheme.

The problem described above as parameter one is an example of a compiler implementation decision which affects the mapping. In effect, one could imagine that the Slim Manufactory produces several models of the Slim computer which differ in the structure of the linkage area, and in the microcode of instructions which manipulate the linkage area (the mark, call and return instructions). Portability of the Remember and Transfer procedures can be maintained by adjusting the manifest constants ECell, CCell, and LinkSize.

The problem described as parameter two of the mapping, occurs, for example, when a PDP-11 is the host machine. Any procedure which makes use of absolute addresses of stack cells - as do Remember, Transfer and Aptovec - must be modified to take account of the direction of stack growth. To aid in maintaining portability in the face of this problem, a manifest constant Forward and a function IsForward are defined as follows:

```
                                    || true:  upward stack
manifest { Forward = false } ||
                                    || false: downward stack
let IsForward(x) = Forward -> x, -x
```

The manifest Forward is true for "upward" growing stacks as on Slim itself, and false for "downward" growing stacks. The function IsForward adjusts a quantity x, usually an offset, according to the stack direction. An example of the use of the function IsForward in Remember would be to note that the value of the E register to be Remember-ed is found by:

```
(@Recover + IfForward(-LinkSize))!ECell
```

since, if the stack grows downward and the order of cells in the linkage area is as described above, the address of the linkage

cell which contains the old value of the E register would be

@Recover + LinkSize

rather than

@Recover - LinkSize.

Portability of the procedures Remember, Transfer, and Aptovec is preserved in the face of this problem by using the IsForward function and by suitably adjusting the manifest constants Forward, ECell and CCell. (See the Appendix for versions of these procedures which have been revised to maintain portability under the three identified mapping parameters. In the revised Aptovec it is assumed that when Forward is false, vectors are implemented in such a fashion that as one traverses the stack in the direction of stack growth from a vector variable v, the first cell of the n + 1 cells of the vector encountered is in fact v!n, and the last one is v!0. Thus in Aptovec, v!0 eventually vecomes v!n of the dynamic vector. This assumption is made so that the BCPL vector subscripting mechanism will work as expected.)

The problem described as parameter three of the mapping also occurs when the PDP-11 is the host machine. The PDP-11 is byte addressable and a single cell of the Slim computer corresponds to two bytes of the PDP-11. A Slim address must be multiplied by 2 in order to get a correct machine address when necessary. If an IBM 360/370 or one of its clones such as an Amdahl 470 were the host, a Slim address would have to be multiplied by 4. Any procedures which require host machine as opposed to Slim addresses must be modified to take care of this problem. In order to maintain portability of such BCPL procedures we define a manifest constant AddressUnitsPerCell and a function MachineAddress as follows:

```
manifest { AddressUnitsPerCell = 2 } || 1 for Slim
                                     || 2 for PDP-11, etc.
let MachineAddress(CellAddress) =
            CellAddress * AddressUnitsPerCell
```

In an implementation of BCPL via Slim on the PDP-11, machine addresses are used in the linkage area. Therefore, in the procedure Transfer, when the OldLink is modified care must be taken to insure that a machine address rather than a cell address is stored in the ECell of the link:

```
OldLink!ECell :=
    MachineAddress(Recover!SaveH+IfForward(LinkSize))
```

The revised procedures in the Appendix make the assumption (following a local implementation of BCPL via Slim on the PDP-11) that machine addresses are required by the link but otherwise BCPL cell addresses can be used. The definition of MachineAddress would have to be altered, of course, if the host machine address of a Slim cell were not simply a multiple of the cell address.

It is believed that the portability of Remember, Transfer and Aptovec, and indeed, of any other BCPL procedure could be maintained should more parameters of the mapping of Slim to a host machine be identified: the method would be to define additional manifest constants and procedures to deal with such new parameters.

More Efficient Portable Procedures

If one allows an inline code facility in BCPL, then it is possible to provide more efficient but still portable versions of some of these procedures. (The measure of efficiency here is the number of abstract machine instructions in the translated version of the procedure.) The BCPL vile command was introduced in [Abramson et al, 1978] with the following syntax and semantics:

vile "string"

The "string" is copied, stripped of its surrounding quotation marks, directly into the stream of abstract machine instructions.

The following version of Aptovec makes use of a single vile command:

```
let Aptovec(f, n) = valof
{ let v = vec 0
  v!0 := n + 1  || stack n
  vile "MH"     || v = vec n
  resultis f(v, n)
}
```

The inserted "MH" is a Modify High Point instruction (mnemonic "M") with an operand (mnemonic "H") which indicates that the

value by which the H register is to be modified is to be found at the top of the stack. This value, in v!0, is popped just before the H register is modified. (The procedures Remember and Transfer given above are based, in fact, on versions written by John Peck which originally used vile commands liberally; they, however, were limited in application only to unconditional transfers of control across procedures.)

It should be emphasized that these extensions of portability by the abstract machine technique are not limited to the Slim machine: portable versions of LEVEL, LONGJUMP, and APTOVEC have been written for the Pica-B and INTCODE machines. Nor are these extensions limited to porting BCPL: they should be applicable to porting any high-level language. Highly machine dependent software can still be portable as long as the dependence is on a carefully defined abstract machine.

A Comment on the Design of BCPL

One final comment about the design of BCPL may be in order. BCPL is quite rich in control structures, so rich in fact, that "most of the time goto-commands are not needed" ([Richards & Whitby-Strevens, 1979]). The procedures Remember and Transfer given above implement all unconditional transfers of control at relatively low costs of time and space. Since goto commands in BCPL are very rarely necessary, and since transfers across procedures must be implemented by procedures anyhow, it seems as if BCPL could easily dispense with the linguistic apparatus of the goto command. Removal of this redundant and little used construct would also have the beneficial effect of simplifying BCPL compilers.

Acknowledgments

This research was supported by the National Science and Engineering Research Council of Canada.

I would like to thank John Peck for a close and critical reading of an early version of this paper, and Gordon Simon, who in porting BCPL via Slim to the PDP-11 provided certain interesting problems to solve.

References

[Abramson et al. 78]
Abramson, H.D. & Fox, Mark & Gorlick, Michael & Manis, Vince & Peck, John, The Pica-B Computer: An Abstract Target Machine for a Transportable Single-User Operating Environment, Proceedings ACM 78, Washington, D.C. December 4-6, 1978.

[Abramson & Peck in preparation]
Abramson, HD. & Peck, J.E.L., The Design of the Slim computer, in preparation, October 1980.

[Fox 78]
Fox, M., Machine architecture and the programming language BCPL, M.Sc. Thesis, Dept. of Computer Science, University of British Columbia, Vancouver, Canada, 1978.

[Richards 71]
Richards, M., The portability of the BCPL compiler, Software - Practice and Experience, vol. 1, 1971.

[Richards 74]
Richards, M., Bootstrapping the BCPL compiler using INTCODE, Machine Oriented Higher Level Languages (van der Poel, Maarsen, editors) North Holland 265-270, 1974.

[Richards & Whitby-Strevens 79]
Richards, M. & Whitby-Strevens, C., BCPL-the language and its compiler, Cambridge University Press, 1979.

Appendix

```
manifest { AddressUnitsPerCell = 1      || 1 for Slim
                                        || 2 for PDP-11
                                        || 4 for 360/370

                                        || true:  upward stack
           Forward             = true   ||
                                        || false: downward stack
           LinkSize            = 2
           ECell               = 0
           CCell               = Forward->1,-1
           SaveE               = 0
           SaveLabel           = 1
           SaveH               = 2
         }

static { NewLink     = 0
         RecoverCopy = 0
       }

let Remember(Recover, Label) = valof
|| save old E, target of Transfer and old H
{ Recover!SaveE := (@Recover+IfForward(-LinkSize))!ECell
  Recover!SaveLabel := Label
  Recover!SaveH := @Recover + IfForward(-LinkSize-1)
  resultis Recover
}

and Transfer(Recover) be
{ let OldLink = @Recover+IfForward(-LinkSize)
  || Create new link just above old H.
  NewLink := Recover!SaveH+IfForward(1)
  || Use static variables to prevent overwriting.
  RecoverCopy := Recover
  || Is this a goto in the same environment?
  if OldLink = NewLink
  do { NewLink!CCell := Recover!SaveLabel
       return
     }
  || Set ECell to old H + LinkSize,
  || adjusting for stack direction and machine addressing:
  OldLink!ECell :=
          MachineAddress(Recover!SaveH+IfForward(LinkSize))
  OldLink!CCell := ReturnHere
  return
ReturnHere:
  || Complete formation of new link and return:
  NewLink!ECell := RecoverCopy!SaveE
  NewLink!CCell := RecoverCopy!SaveLabel
}
```

```
and MachineAddress(CellAddress) =
            CellAddress * AddressUnitsPerCell

and IfForward(x) = Forward -> x, -x


and Aptovec(f, n) = valof
{ let Save = vec 2
  and v    = vec 0
  Remember(Save, LabelA)
  || Modify saved value of H register,
  || adjust for stack direction:
  Save!SaveH +:= IfForward(n)
  unless Forward do v -:= n
  Transfer(Save)          || v = vec n
LabelA:
  resultis f(v, n)
}

and AptovecVile(f, n) = valof
{ let v = vec 0
  v!0 := n + 1            || stack n + 1
  vile "MH"               || v = vec n
  unless Forward do v -:= n
  resultis f(v, n)
}
```