Volume 4 Number 1

January 1986

J. Mende  Research Directions in Information Systems  1
R. Dempster  The SECD Machine: an Introduction  5
J.M. Bishop  Ways of Assessing Programming Skills  11
N.C. Phillips  An Alternative Development of Vienna Data Structures  21

Book Reviews  20

An official publication of the Computer Society of South Africa and of the South African Institute of Computer Scientists

'n Amptelike tydskrif van die Rekenaarvereniging van Suid-Afrika en van die Suid-Afrikaanse Instituut van Rekenaarwetenskaplikes
An official publication of the Computer Society of South Africa and of the South African Institute of Computer Scientists

Editor
Professor G. Wiechers
INFOPLAN
Private Bag 3002
Monument Park 0105

Editorial Advisory Board
Professor D.W. Barron
Department of Mathematics
The University
Southampton S09 5NH, England

Professor J.M. Bishop
Department of Computer Science
University of the Witwatersrand
1 Jan Smuts Avenue
Johannesburg 2001

Professor K. MacGregor
Department of Computer Science
University of Cape Town
Private Bag
Rondebosch 7700

Dr H. Messerschmidt
IBM South Africa
P.O. Box 1419
Johannesburg 2000

Subscriptions
Annual subscriptions are as follows:

<table>
<thead>
<tr>
<th></th>
<th>SA</th>
<th>US</th>
<th>UK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Individuals</td>
<td>R10</td>
<td>$7</td>
<td>£5</td>
</tr>
<tr>
<td>Institutions</td>
<td>R15</td>
<td>$14</td>
<td>£10</td>
</tr>
</tbody>
</table>

Quaestiones Informaticae is prepared by the Computer Science Department of the University of the Witwatersrand and printed by Printed Matter, for the Computer Society of South Africa and the South African Institute of Computer Scientists.
The SECD Machine: An Introduction

ROBERT DEMPSTER
Department of Computer Science
University of Natal, Pietermaritzburg
3200

INTRODUCTION

The S(tack), E(nvironment), C(ontrol) and D(ump) machine was invented by Landin (1964) to perform the mechanical evaluation of symbolic expressions. The evaluation of an expression invariably involves the application of an operator to an operand. The concept of an applicative expression (AE) is thus formally introduced and discussed. This is followed by an overview of the SECD machine and a detailed trace of the SECD machine state transformations during the evaluation of an AE on the SECD machine.

APPLICATIVE EXPRESSIONS

Landin (1964) defines an AE as either

- an identifier
- or a λ-expression (λexp) which has a bound variable (bv) which is an identifier or
  identifier list, and a λ-body (body) which is an AE,
- or a combination which has an operator (rator) which is an AE, and an operand
  (rand) which is an AE.

Identifiers comprise single- and multi-character constants and variables. The decimal numbers are also considered to be identifiers and as such they represent single entities viz. the value of the decimal numbers.

Examples of identifiers are trivial except that, as will be seen later, such identifiers are bound to values within a certain environment. As such they may have simple values or may in turn represent AEs.

λ-expressions can be illustrated by drawing parallels with conventional high school mathematical functions, e.g

\[ f(x) = x^2 + 2 \quad f = \lambda x \cdot x^2 + 2 \]
\[ g(x,y) = x^2 + y^2 \quad g = \lambda (x,y) \cdot x^2 + y^2 \]

The notation used places the bv part between the λ and the dot which is immediately followed by the λ-body.

In the context of a computation AEs may appear in the following forms.

As an identifier the AE represents some value which will be determined by the environment in which the identifier is bound. This value may be a primitive i.e. a constant of the environment or another AE.

As a lambda expression the AE represents functional abstraction. This form of AE will always denote something provided we know the value of each variable that occurs free in it. For example

\[ \lambda x \cdot x^2 + y^2 + 2 \]

will be valid provided the free variables y and 2 can be bound to specific values. It is worth noting here, that "2" is treated as the numeral "2" bound in some environment to the integer 2. Also that other identifiers such as square, sin etc. may exist bound in the same environment to basic functions which are effectively applied in one step.

Thirdly the AE expression may occur in the form of a functional application or operator/operand combination. Here both the operator (rator) and operand (rand) are in turn AEs and as such the rator and rand will have to be evaluated before the former can be applied to the latter. It is worth noting that mathematics makes no assumptions regarding the order in which we evaluate the rator and rand of a functional application.

To make sense the rator, if it is not an identifier bound to some function, must then be an AE which will evaluate to a function. The rand must either be an identifier bound to a value within the domain of the rator or it must be an AE which will evaluate to some such value.
From this we will gather that AEs are a type of composite information structure. In order to process such information structures, we require various operations. Landin (1964) suggests that we will require predicates for testing for alternative AE formats as well as selectors and constructors for partitioning and building AEs respectively. Typical examples are:

\[
\begin{align*}
\text{is-\lambda exp} (\lambda x. x + 2) &= \text{true} \\
\text{sel-bv} (\lambda x. x + 2) &= x \\
\text{constr-\lambda exp} (\text{sel-bv}(X), \text{sel-body}(X)) &= X
\end{align*}
\]

where \(X\) is a \(\lambda\)-expression.

In the earlier definition of AEs the bv of a \(\lambda\)-expression was described as an identifier or an identifier list. The texts referenced for this paper tended to use \(\lambda\)-expressions with a single bound variable. This is because it is possible to replace a function of several variables by a complex function of only one variable, e.g.

\[
\lambda (x, y). x + y 
\rightarrow \lambda x. \lambda y. x + y
\]

Although more complex the latter representation is more suited to the mechanical evaluation process as performed by the SECD machine.

AE EVALUATION

To be useful in the context of computing, an AE will ultimately have to take on some value. This value may either be a number or a function or a list of numbers or functions. In order to evaluate an AE we will have to be able to determine the value of any variables occurring free in the AE. Thus we must associate with the AE an environment (E) which contains the necessary free variables appropriately bound. This environment is readily implemented in terms of a structure consisting of name/value pairs. We thus associate with the evaluation of an AE a closure which consists of the AE and its appropriate environment. The construction operation constr-clos(\(\lambda\)exp, E) does just that while is-clos is a predicate for testing for the presence of a closure. The selectors clos-body, clos-bv and clos-env select the respective parts of the closure in question.

STACKS

As stacks are crucial to an appreciation of the SECD machine a brief discussion of the stack structure and operations employed follows.

The stacks are implemented as linked lists and can thus be processed using the usual list and stack operations. Consider an empty or null stack \(S = []\). After the push operations push \((z, S)\), push \((y, S)\) and push \((x, S)\) we have \(S = [x, y, z]\) with the element \(x\) on the top of the stack. A notational device `'::' is used to denote the result of a push operation and in this context push \((w, S)\) will result in a new stack value represented by \(w :: S\) i.e. \([w, x, y, z]\).

In terms of the linked lists underlying the stacks we also employ the selectors head \((h)\) which selects the element on the top of the stack, tail \((t)\) which effectively selects the popped stack and then 1st, 2nd etc. which select the first, second etc. elements from the stack respectively.

Using the stack \(S\) described above

\[
\begin{align*}
\text{h} & \quad S = w \\
\text{t} & \quad S = [x, y, z] \\
\text{1st} & \quad S = w \\
\text{2nd} & \quad S = x \quad \text{etc.}
\end{align*}
\]

A final note regarding the stacks is that as the stack elements are variant they must thus be tagged in order to allow the type of any element in question to be determined. Suitable predicates are employed for this purpose.

THE SECD MACHINE

Like all other computing machines the SECD machine proceeds through a number of states as it evaluates an AE. The SECD machine state is represented by a 4-tuple of stacks i.e. \(<S, E, C, D>\). \(S\) is an evaluation stack which is used for computation in terms of the application of a basic function on the top of the stack to an operand(s) immediately beneath it. The result replaces the operator and operand(s) on the top of the stack. \(E\) is a stack used to store the current
environment. It consists of a number of name/value pairs and can also be considered as an association list. C is the control stack and it initially contains the AE which is to be evaluated. D is the dump and it is used to store the current state of the SECD machine each time a nested function application takes place. This is done by pushing the current state onto D and then initializing S, E, and C in terms of the nested function application. After the nested function application is complete the previous SECD state is restored by popping it off D.

The SECD machine is conceptually not unlike conventional von Neumann machines in the context of the latter's execution of programs written in procedural languages such as PASCAL. When the code for these programs is executed a stack frame is created on the stack for each procedure evocation. The stack frame is typically built up from the address to which control must return after the procedure terminates, the procedure parameter list, and the list of variables declared local to the procedure. When the procedure terminates the stack frame is removed from the stack exposing the stack frame associated with the calling procedure. If the procedure was typed, (i.e. a function) then the value returned by the function would have been left on the top of the stack. The dynamic nesting of procedure calls is thus captured on the stack. This situation does not however necessarily reflect the static nesting or scope of the currently evoked procedure. To keep track of the static environment of the currently evoked procedure a set of display registers is used. These describe the static environment by means of pointers to the relevant stack frames and are used to resolve the visibility of variables within this environment.

In this context the 'stack' of the SECD machine is the current stack frame, the 'environment' is the static scope in terms of the stack frames pointed to by the display registers, the 'control' is the code and the 'dump' is the current stack up to and including the previous dynamically linked stack frame.

Having illustrated these similarities it is interesting to compare AEs with their equivalent PASCAL code segments.

1. Function \( f \) (\( x : \text{real} \)) : \text{real};
   begin
   \( f := x \times x; \)
   end;

\( f \ (3) = \{\lambda x. x \times x\} \ (3) = 9 \)

2. Function \( g \) (Function \( h : \text{real}; \ x : \text{integer} \)) : \text{real};
   begin
   \( g := h(x) + h(x); \)
   end;

\( g \ (\text{sin}, \ 3) = \{\lambda h. \lambda x. h(x) + h(x)\} \ (\text{sin}, \ 3) = \{\lambda x. \text{sin}(x) + \text{sin}(x)\} \ (3) = \text{sin}(3) + \text{sin}(3) = 0.28224 \)

3. Function \( g \) (\( y : \text{real} \)) : \text{real};
   Function \( f \) (\( x : \text{real} \)) : \text{real};
   begin
   \( f := x \times x; \)
   end;
   begin
   \( g := f(y) + f(y); \)
   end;

\( g \ (2) = \{\lambda y. \{\lambda f.f(y) + f(y)\} (\lambda x. x \times x)\} \ (2) = \{\lambda y. (\lambda x. x \times x) (y) + (\lambda x. x \times x) (y)\} \ (2) = (\lambda x. x \times x) \ (2) + (\lambda x. x \times x) \ (2) = 4 + 4 = 8 \)
The preceding examples suggest that the constructs of a language such as PASCAL could be modelled in terms of AEs. These AEs could then be evaluated on the SECD machine to determine their outcome and thus the meaning or semantics of the PASCAL constructs. This approach to the formal semantics of a programming language is called operational in terms of the interpretation of the language's constructs by an abstract machine.

This method provides a further tool in that the interpreter itself could easily be altered in order to study the effect this will have on the semantics of the language in question. The most obvious example of such an application would be an investigation of the different mechanisms for passing arguments to functions i.e. by value, reference or name.

THE SECD MACHINE MECHANISM

In order to evaluate an AE using the SECD machine it must first be initialised in terms of a null evaluation stack S (hereafter referred to as the stack), an environment e.g. $E = \{ x . 2, y . 4, '5' . 5 \}$ i.e. the names $x$, $y$ and '5' bound to the integers 2, 4 and 5 respectively, an AE e.g. $C = \{ \{ \lambda . x + y + z \} (5) \}$ and a null dump $D$.

Brady (1977) proposes that the initialisation process be carried out by a function $loadAE$ which loads an AE, $C$, and its corresponding environment $E$ into an SECD machine state. The following function could be used:

$$loadAE(C, E) = iter(transform, <\[, E, C, \[\]>)$$

where $iter$ is a recursive function that calls $transform$, the SECD machine interpreter. These functions will be described shortly.

Once the SECD machine has been initialised $iter$ recursively calls $transform$ which interprets the control and so doing advances the SECD machine to its next state. In this manner the machine proceeds through a number of SECD states driven by the control until the latter and the dump are both null. The head of the stack should then contain the value of the AE.

If the control is null and the dump is not null then a nested AE has just been evaluated and the dump must thus be popped. This is achieved by replacing the stack with the stack value that is on the dump with the head of the current stack pushed onto it. The latter corresponds to a value returned as a result of the evaluation of the nested AE. The tail of the current stack if it exists is lost. $E$ and $C$ are replaced by their respective values from the state which is on the dump. The current dump, $D$ is replaced by the previous dump. In effect the previous dump is popped. The new control list is then interpreted.

An examination of the function $transform$ will reveal that it only advances the SECD machine from the current state to the next. In order to continue the transformation process the function $iter$ is recursive. The SECD machine state transitions described above are implemented by $iter$ as shown below.

The notation employed to code these algorithms is based on McCarthy's formalism as described by Brady (1977). A set $F$ of base or primitive functions, and a set $C$ of functionals for building new functions out of old is assumed. The closure $C(F)$ should then consist of all the computable functions in terms of the choice of $F$ and $C$.

The set $F$ can be devised in accordance with the nature of the set of problems which has to be solved i.e. numerical, textual, etc. The set $C$ however is more specific and the set proposed by McCarthy is used viz. generalised composition of functions, conditional expressions and recursion. We note that the PASCAL conditional expression

$$\text{if } p_1 \text{ then } e_1$$
$$\text{else if } p_2 \text{ then } e_2;$$

is given by McCarthy as

$$(p_1 \rightarrow e_1, p_2 \rightarrow e_2, \ldots, p_n \rightarrow e_n)$$

where $p_n$ is often left out if $p_n$ is true.

$$iter \ (trans, state) =$$

$$\begin{cases} (null \ (C) \rightarrow) \\ (null \ (D) \rightarrow ()), /* \text{result on the stack evaluation complete} */ \\ (S = h(S) :: S', /* \text{pop the previous state from the current D} */ \\ E = E' \\ C = C' \\ D = D') \end{cases}$$
The interpretation of the control as performed by transform entails determining the value of the head of the control. If the head of the control list is:

1. an identifier then the value of the identifier is found by searching the current environment using the function lookup. The value found is pushed onto the top of the current stack and the control is replaced by its tail.
2. a λexp then a closure is constructed in terms of the current environment and the λexp and this value is pushed onto the stack. The control is replaced by its tail.
3. a combination i.e. rator/rand pair then the combination is popped off the control and the ap, the rator and the rand are pushed back onto the control in that order.
4. ap, a special object distinct from all AEs, then if:
   - the value which is at the top of the stack is a basic function (i.e. one which can be executed atomically) it is applied to the 2nd element of the stack. The result replaces the topmost two elements of the stack.
   - the value which is at the top of the stack is a closure then a new environment is created by prefixing the clos-env with the actual argument (2nd S) bound to the formal argument (clos-bv). The control is set to the value of the clos-body. The stack is cleared while the dump takes on the value of the SECD state just used to create the new one.

The underlying interpretation of the control is carried out by a function transform devised by Landin (1964). The version of transform given below is largely due to Brady (1977).

```
transform (S, E, C, D) =
(let x = h (C)
    is-ident (x) → <lookup (E, x) :: S, E, t (C), D>,
    is-comb (x) → <S, E, C', D>
    where C' = sel-opd(x):: (sel-opr(x):: (ap: :t (C))),
    is-lexp (x) → <make-clos (x, E) :: S, E, t (C), D>,
    X = ap →
    (let f = h (S)
        not is-clos (f) → <S', E, t (C), D>
        where S' = f (2nd (S)) :: t (t (S))),
    (let cl-body = sel-body (f)
    let cl-bv = sel-bv (f)
    let cl-env = sel-env (f)
    []), extend (cl-env, 2nd (S), cl-bv), cl-body, D'>
    where D' = < t (t (S)), E, t (C), D>
))
```

A SIMPLE EXAMPLE

Consider the AE \{λz. x + y + z\} (5) in the environment \{x. 2, y. 4, '5'.5\} i.e. the identifiers x, y and '5' bound to the integers 2, 4 and 5 respectively.

For the purposes of evaluation we convert the infix expression to its prefix equivalent. We also introduce '+i' as a one argument free variable which when applied to an integer i yields the function (+i). When (+i) is applied to an integer j, the integer, (i+j) is obtained. These transformations are described by Wegner (1968).

The call loadAE \{{λz. ((+(x) y)) z\} (5), [x.2, y.4, '5', 5]\) results in the following SECD machine state trace:

```
<S, E, C, D>
<[], [x.2, y.4, '5', 5], [[{λz. ((+(x) y)) z} (5)]]>, []>
```
/* hC is a combination i.e. rator/rand pair */
<[], [x.2, y.4, '5', 5], [], [[λz. (+((+x) y)) z]], ap], []>
/* hC is an identifier */
<[], [x.2, y.4, '5', 5], [[λz. (+((+x) y)) z]], ap], []>
/* hC is a λ-expression, so make closA */
<[], [x.2, y.4, '5', 5], [[ap]], []>
/* closA contains {λz.(+((+x)y))z} and E, hC is ap so prepare to evaluate the closure by extending E with the cl-bv z bound to arg 5, set C = cl-body and push rest of the SECD state onto D = <[], [x.2, y.4, '5', 5], [], []>
<[], [z.5, x.2, y.4, '5', 5], [[(+((+x)y))z]], D>
/* hC is a combination i.e. (+((x)y)) applied to z */
<[], E, [z, (+((+x)y))], ap], D>
<[], E, [+((+x)y)], ap], D>
<[], E, [((+x)y), +, ap, ap], D>
<[], E, [y, (+x), ap, +, ap, ap], D>
<[], E, [x, ap, ap, +, ap, ap], D>
/* hC is a lambda expression which is basic function */
<[], E, [ap, ap, +, ap, ap], D>
/* apply 1st S to 2nd S */
<[], E, [ap, +, ap, ap], D>
/* apply 1st S to 2nd S */
<[], E, [+ap, ap], D>
/* the next three steps as before */
<[], E, [ap, ap], D>
<[], E, [ap], D>
<[], E, [], []>
/* C is null so pop D */
<[], [x.2, y.4, '5', 5], [], []>
/* C and D both null, finish result on S */

CONCLUSION

As claimed by the title this work is intended as an introduction to the SECD machine and its use to evaluate applicative expressions. Hopefully the discussion and accompanying example will in some way fill in some of the gaps which I experienced when first coming to grips with the available literature. The work is also intended to serve as a basis for further study allied to AE's and the SECD machine.

ACKNOWLEDGEMENTS

I would like to thank Prof. S.W. Postma for suggesting the topic and for time spent reviewing the work.

REFERENCES

The purpose of this journal will be to publish original papers in any field of computing. Papers submitted may be research articles, review articles, exploratory articles of general interest to readers of the Journal. The preferred languages of the Journal will be the congress languages of IFIP although papers in other languages will not be precluded.

Manuscripts should be submitted in triplicate to:
Prof. G. Wiechers at:
INFOPLAN
Private Bag 3002
Monument Park 0105

Form of manuscript

Manuscripts should be in double-space typing on one side only of sheets of A4 size with wide margins. The original ribbon copy of the typed manuscript should be submitted. Authors should write concisely.

The first page should include the article title (which should be brief), the author's name, and the affiliation and address. Each paper must be accompanied by an abstract less than 200 words which will be printed immediately below the title at the beginning of the paper, together with an appropriate key word list and a list of relevant Computing Review categories.

Tables and figures

Illustrations and tables should not be included in the text, although the author should indicate the desired location of each in the printed text. Tables should be typed on separate sheets and should be numbered consecutively and titled.

Illustrations should also be supplied on separate sheets, and each should be clearly identified on the back in pencil with the Author's name and figure number. Original line drawings (not photoprints) should be submitted and should include all relevant details. Drawings etc., should be submitted and should include all relevant details. Drawings etc., should be about twice the final size required and lettering must be clear and open and sufficiently large to permit the necessary reduction of size in block-making.

Where photographs are submitted, glossy bromide prints are required. If words or numbers are to appear on a photograph, two prints should be sent, the lettering being clearly indicated on one print only. Computer programs or output should be given on clear original printouts and preferably not on lined paper so that they can be reproduced photographically.

Figure legends should be typed on a separate sheet and placed at the end of the manuscript.

Symbols

Mathematical and other symbols may be either handwritten or typewritten. Greek letters and unusual symbols should be identified in the margin. Distinction should be made between capital and lower case letters; between the letter Q and zero; between the letter I, the number one and prime; between K and kappa.

References

References should be listed at the end of the manuscript in alphabetical order of author's name, and cited in the text by number in square brackets. Journal references should be arranged thus:


Proofs and reprints

Galley proofs will be sent to the author to ensure that the papers have been correctly set up in type and not for the addition of new material or amendment of texts. Excessive alterations may have to be disallowed or the cost charged to the author. Corrected galley proofs, together with the original typescript, must be returned to the editor within three days to minimize the risk of the author's contribution having to be held over to a later issue.

Only original papers will be accepted, and copyright in published papers will be vested in the publisher.

Letters

A section of "Letters to the Editor" (each limited to about 500 words) will provide a forum for discussion of recent problems.

Hierdie notas is ook in Afrikaans verkrygbaar.