A New Implementation Technique for Applicative Languages

D. A. TURNER
Computer Laboratory, University of Kent, Canterbury CT2 7NF

SUMMARY

It is shown how by using results from combinatory logic an applicative language, such as LISP, can be translated into a form from which all bound variables have been removed.

Aside from its use in defining the semantics of languages with bound variables, the 'environments' model can fairly be considered the basis of all the conventional methods of implementing this language feature. The apparently wide divergences between one implementation technique and another are largely due to the choice of different concrete data structures to represent the abstract data type 'environment'. The representation of the environment generally used in interpreters is a linked list of name-value pairs (an 'association list'), see for example Landin's SECD machine[2] and of course the standard LISP interpreter.[3] Another strategy, used by most Algol compilers, is to keep variables on the stack and represent the environment by a small data structure, the 'display',[4] which gives access to various reference points on the stack. These are the two commonest strategies though other variations occur.

A radically different method of dealing with bound variables is possible, however, at least for purely applicative languages (those without assignment or side effects) and it is this that we describe here. It does not derive from the 'environments' model at all but rather from a curious result in logic which shows that variables, as they are used in logic and ordinary mathematics are not strictly necessary.[5] Given a modest number of extra constants, called *combinators*, we can systematically translate whatever we have to say into a notation in which bound variables do not occur. This process of removing variables can be thought of as a kind of compilation and the resulting variable-free notation as a kind of object code. Although it is quite unreadable by human beings this code can be efficiently 'executed' by a machine of a very simple character. The author has found that it is possible along these lines to produce a viable implementation of an applicative language such as pure LISP (in fact SASL was used). It was found to be broadly comparable in speed and space utilization to a conventional interpreter using association lists but to have certain advantages.

A number of recent authors[10–12] have advocated that the semantics of applicative languages should be redefined so as to permit *non-strict functions*. (A non-strict function is one that can return an answer even if one of its arguments is undefined.) In a conventional implementation the necessary changes are found to bring about a slowdown in execution speed of up to an order of magnitude. The implementation technique described here, however, provides this behaviour quite naturally and without any extra cost. The second principal advantage is that the combinatory 'code' turns out to have some remarkable self-optimizing properties including that constant calculations are automatically moved outside loops and that the overhead cost of calling a user-defined function falls to zero after the first occasion of its use.

The remainder of the paper is organized as follows. In the first section we discuss the algorithm used for removing bound variables from the source text ('compiling'). In the second section we outline the strategy of the machine which executes the resulting code. Finally, in the third section we give some preliminary performance figures and discuss the properties of this kind of implementation. In order to make the article self-contained absolutely no technical knowledge of combinatory logic on the part of the reader has been assumed. The results developed in the first section are in fact well known and can be found in the standard text.[6]

REMOVING VARIABLES FROM THE SOURCE TEXT

We start by introducing a notion crucial to the whole plan—that of a higher order function. An ordinary function, say *sin*, returns for its result a number or similar simple object. But it is possible to conceive of a function which returns for its result another function.

The 'differentiate' operation of school calculus is a function of this kind, for example we might write

$$D\ sin = cos$$

showing that the (higher order) function D when applied to the argument *sin* returns for its result the function *cos*. Note the convention here that the application of a function to its argument is denoted simply by juxtaposition—we do not insist on enclosing the argument in brackets. We will further assume that juxtaposition associates to the left, so we can write, say

$$D\ sin\ 0 = 1$$

meaning

$$(D\ sin)\ 0 = 1, \quad \text{i.e. } cos\ 0 = 1$$

As an additional example of a higher order function consider the following definition of a function *plus*

$$plus\ x\ y = x+y$$

meaning that *plus* can be applied to an argument, x say, and returns a function which when applied to an argument, y say, returns the sum of x and y. So

$$plus\ 2\ 3 = 5$$

but it is read as '(*plus* 2) 3'. And '*plus* 2' has a meaning in its own right—it is the function that adds two to things. (This device for reducing a first order function of several arguments to a higher order function of one argument is called *currying* after the logician H. B. Curry. Thus we would say that *plus* is here a curried version of the + operation.)

The basic algorithm

We are now ready to begin removing variables from the source text. The source is in the form of a series of messages from the user to the SASL system (the system is interactive). Each message is either an expression to be evaluated, in which case its value is printed immediately, or else it is a definition to be stored for latter use, like

$$\textbf{def}\ pi = 3{\cdot}141592$$

or like the definition of factorial given earlier.

To begin with a very simple example take the following definition of the successor function

$$\textbf{def}\ suc\ x = x+1 \tag{1}$$

The aim is to eliminate the variable x obtaining a definition of the form

$$\textbf{def}\ suc = \ldots$$

where the right hand side is an expression containing only constants. The first step is to rewrite equation (1) using a curried version of the + operator

$$\textbf{def}\ suc\ x = \textbf{plus}\ 1\ x \tag{2}$$

Now we can remove x from both sides of the equation obtaining

$$\textbf{def}\ suc = \textbf{plus}\ 1 \tag{3}$$

which is an acceptable solution. (A good question is, why can we 'cancel' x in this way? The anwer is, because of the *principle of extensionality* which states that two functions f and g say, are equal if and only if: $f\ x = g\ x$ for all x.)

The step from equation (2) to equation (3) was very easy in this case because the variable to be removed from the body of the function occurred only once, at the extreme right. To handle the general case we need to borrow some technical results from combinatory logic. We take as the typical definition

$$\textbf{def}\, f\, x = \ldots \tag{4}$$

where ... is an expression built up from constants and the variable x using various operators. For a first step we replace all the operators by their curried versions, giving

$$\textbf{def}\, f\, x = E \tag{5}$$

where E is an expression in which functional application is the only operation (such an expression is called a *combination*). We can now write the solution as

$$\textbf{def}\, f = [x]\ E \tag{6}$$

where $[x]\ E$ denotes the result of a textual operation to be defined shortly, pronounced 'abstract x from E' and which removes all occurrences of x. For equation (6) to be a correct solution we require that

$$([x]\ E)\ x = E \quad \text{(law of abstraction)}$$

i.e. that abstraction be an exact inverse of application. We introduce three combinators, **S**, **K** and **I**, defined by the equations

$$\mathbf{S}\, f\, g\, x = f\, x\, (g\, x) \tag{S}$$

$$\mathbf{K}\, x\, y = x \tag{K}$$

$$\mathbf{I}\, x = x \tag{I}$$

and define the abstraction operation as follows

$$[x]\ (E_1\ E_2) \Rightarrow \mathbf{S}\ ([x]\ E_1)\ ([x]\ E_2)$$

$$[x]\ x \Rightarrow \mathbf{I}$$

$$[x]\ y \Rightarrow \mathbf{K}\ y$$

where y is a constant or a variable other than x. That abstraction thus defined obeys the law of abstraction given above can be proved by an induction on the size of the combination E—we leave this as an exercise for the interested reader.

This is the basic algorithm for removing variables but if used without modification it tends to produce rather long-winded 'code'. For the successor function we get

$$\begin{aligned}\textbf{def}\ \mathit{suc} &= [x]\ (\textbf{plus}\ 1\ x)\\ &\Rightarrow \mathbf{S}\ ([x]\ (\textbf{plus}\ 1))\ ([x]\ x)\\ &\Rightarrow \mathbf{S}\ (\mathbf{S}\ (\mathbf{K}\ \textbf{plus})\ (\mathbf{K}\ 1))\ \mathbf{I}\end{aligned}$$

which while perfectly correct (as the reader may confirm by applying the above expression to an arbitrary x and checking that the result simplifies, by the use of the equations (S), (K) and (I), to '**plus** 1 x') is much less compact than the '**plus** 1' which we obtained earlier.

To give one more example, from the definition of factorial given at the very beginning of this paper the basic algorithm yields the solution

$$\begin{aligned}\textbf{def}\ \mathit{fac} = \mathbf{S}\ &(\mathbf{S}\ (\mathbf{S}\ (\mathbf{K}\ \textbf{cond})\ (\mathbf{S}\ (\mathbf{S}\ (\mathbf{K}\ \textbf{eq})\ (\mathbf{K}\ 0))\ \mathbf{I}))\\ &(\mathbf{K}\ 1))\ (\mathbf{S}\ (\mathbf{S}\ (\mathbf{K}\ \textbf{times})\ \mathbf{I})\ (\mathbf{S}\ (\mathbf{K}\ \mathit{fac})\\ &(\mathbf{S}\ (\mathbf{S}\ (\mathbf{K}\ \textbf{minus})\ \mathbf{I})\ (\mathbf{K}\ 1))))\end{aligned}$$

where the constants **cond**, **eq**, **times** and **minus** are the curried versions of the conditional, $=$, $\times$ and $-$ operators respectively.

Improving the algorithm

To improve the performance of the algorithm we introduce the extra combinators **B** and **C**, defined by

$$\mathbf{B}\, f\, g\, x = f\, (g\, x)$$

$$\mathbf{C}\, f\, g\, x = f\, x\, g$$

and the optimizations (E_1 and E_2 stand for arbitrary combinations):

$$\mathbf{S}\ (\mathbf{K}\ E_1)\ (\mathbf{K}\ E_2) \Rightarrow \mathbf{K}\ (E_1\ E_2)$$

$$\mathbf{S}\ (\mathbf{K}\ E_1)\ \mathbf{I} \Rightarrow E_1$$

$$\mathbf{S}\ (\mathbf{K}\ E_1)\ E_2 \Rightarrow \mathbf{B}\ E_1\ E_2 \quad \text{if no earlier rule applies}$$

$$\mathbf{S}\ E_1\ (\mathbf{K}\ E_2) \Rightarrow \mathbf{C}\ E_1\ E_2 \quad \text{if no earlier rule applies}$$

(The interested reader can satisfy himself that the left and right hand sides of each of these rules are always equal by applying both sides to an arbitrary x and simplifying.)

The above rules recognize various special cases where the variable being abstracted is absent from one or more subexpressions. This brings about a considerable improvement in the quality of the code produced. The code above for *suc* reduces to **plus** 1 under the above rules and for *fac* we get the much shortened version

def *fac* = **S** (**C** (**B cond** (**eq** 0)) 1) (**S times** (**B** *fac* (**C minus** 1)))

In the author's compiler these optimizations are incorporated as an integral part of the abstraction algorithm and the long-winded form of the code never comes into existence.

Up to now we have been assuming that we have to deal only with definitions of functions of one argument and with no local definitions in the body of the function. In fact the abstraction algorithm can be applied repeatedly to cope with more complex situations. In SASL functions of several arguments are normally defined as curried functions, for example

$$\mathbf{def}\, f\, x\, y = E$$

which would compile to

$$\mathbf{def}\, f = [x]\ ([y]\ E)$$

where the inner abstraction must of course be performed first.

Local variables, introduced by a **where** clause as in

$$E_1 \quad \mathbf{where}\ x = E_2$$

are removed by transforming an expression of the above form to

$$([x]\ E_1)\ E_2$$

So for example the expression

$$(x+1) \times (x-1) \quad \mathbf{where}\ x = 7$$

would compile to

$$\mathbf{S}\ (\mathbf{B\ times}\ (\mathbf{C\ plus}\ 1))\ (\mathbf{C\ minus}\ 1)\ 7$$

This transformation is applied repeatedly to handle arbitrary nestings of **where** inside **where** and inside function bodies—as always inner abstractions are performed first.

Functions can be defined in **where** clauses also but this presents no new difficulties. For example

$$\ldots f \ldots \quad \textbf{where } f\, x = E$$

would compile to

$$([f]\ (\ldots f \ldots))\ ([x]\ E)$$

Data structures

Like LISP, SASL has only one method of data structuring namely a pairing operation. We represent this by an infix colon, thus

$$x : y$$

is the data structure whose head is x and whose tail is y; in LISP it would be written '(*cons x y*)'. As usual this operation can be cascaded to the right to form a linked list and a syntactic sugaring is provided for this. So, for example, a three list is written

$$a, b, c$$

which is taken as shorthand for

$$a : (b : (c : \textbf{nil}))$$

We introduce the combinator **P** as a curried version of the pairing operation and so the above expression compiles to

$$\textbf{P}\ a\ (\textbf{P}\ b\ (\textbf{P}\ c\ \textbf{nil}))$$

In SASL the explicit use of the selectors **hd** and **tl** is generally avoided by using colons and commas on the left of definitions instead. For example it is permitted to write

$$\textbf{def } a : b = x$$

The compiler treats this as equivalent to

$$\textbf{def } a = \textbf{hd}\ x$$

$$\textbf{def } b = \textbf{tl}\ x$$

It is permitted to write arbitrarily complicated 'templates' on the left of a definition in this way. Such templates can also occur on the left of **where** clauses and in formal parameter positions. To compile such constructions the author has introduced the new combinator **U** (for 'uncurry') defined by the equation

$$\textbf{U}\ f\ (x : y) = f\ x\ y$$

and has generalized the abstraction algorithm to permit abstraction with respect to a template. We define

$$[x : y]\ E$$

to mean

$$\textbf{U}\ ([x]\ ([y]\ E))$$

By recursive application of the above rule we can abstract with respect to an arbitrary template (we leave it to the reader to check that this extended definition of abstraction satisfies a suitably generalized version of the law of abstraction). So for example

$$E_1 \quad \textbf{where } a : b = E_2$$

compiles to

$$([a : b]\ E_1)\ E_2 \Rightarrow \mathbf{U}\ ([a]\ ([b]\ E_1))\ E_2$$

and

$$\mathbf{def}\ f\ (x, y, z) = E$$

compiles to

$$\begin{aligned}\mathbf{def}\ f &= [x, y, z]\ E\\ &\Rightarrow [x : (y : (z : \mathbf{nil}))]\ E\\ &\Rightarrow \mathbf{U}\ ([x]\ (\mathbf{U}\ ([y]\ (\mathbf{U}\ ([z]\ (\mathbf{K}\ E))))))\end{aligned}$$

(A detail—in the actual implementation rather than **K** we would here use a combinator with a similar action but which checks that its second argument does in fact take the value **nil**.)

Assembling the code

Using the foregoing rules then, the compiler is able to transform each incoming expression into a pure combination and each incoming **def** definition into one or more definitions of the form

def *name* = *combination*

The combinations are stored as binary trees, with the nodes representing functional applications. At the leaves of the tree there will be constants (like 1 or **plus** or **S**). When an unbound variable occurs it is replaced by the combination with which it has been associated by a **def** definition, or it is so replaced as soon as such a definition is encountered. That is, outer level names (i.e. those defined by **def**) are handled by substituting their definitions as subtrees into the trees in which they occur. Subtrees are included by pointing not by copying, so combinations can share subtrees (since we have no assignment this is perfectly safe). A recursive definition will compile to a cyclic graph. For example, the definition of the factorial function given in the introduction will result in the name *fac* being associated with the graph shown in Figure 1. A subsequent request to evaluate, say *fac* 3, will compile to a tree whose left subtree is the structure shown in Figure 1 and whose right subtree is 3.

(This seems a suitable place to make a comment on the conventions used in diagrams such as that in Figure 1. Letters and numbers will sometimes be drawn inside cells as here and sometimes drawn outside with a pointer to them shown originating from a field of the cell. There is absolutely no logical significance in our using sometimes the one convention and sometimes the other—it is dictated purely by what is topologically convenient when drawing the diagram. Whether, say, numbers are actually stored immediately or stored elsewhere and pointed to, is a low level implementation detail depending on, for example, the word length of the computer, with which this article is not concerned.)

Local recursion

There is one further complication which the compiler must be able to handle—local recursion, as in

$$E_1 \quad \mathbf{where}\ x = \ldots\ x\ \ldots$$

This is transformed to

$$([x]\ E_1)\ (\mathbf{Y}\ ([x]\ (\ldots\ x\ \ldots)))$$

where **Y** is the fixpoint combinator, defined by the equation

$$\mathbf{Y}\ f = f\ (\mathbf{Y}\ f)$$

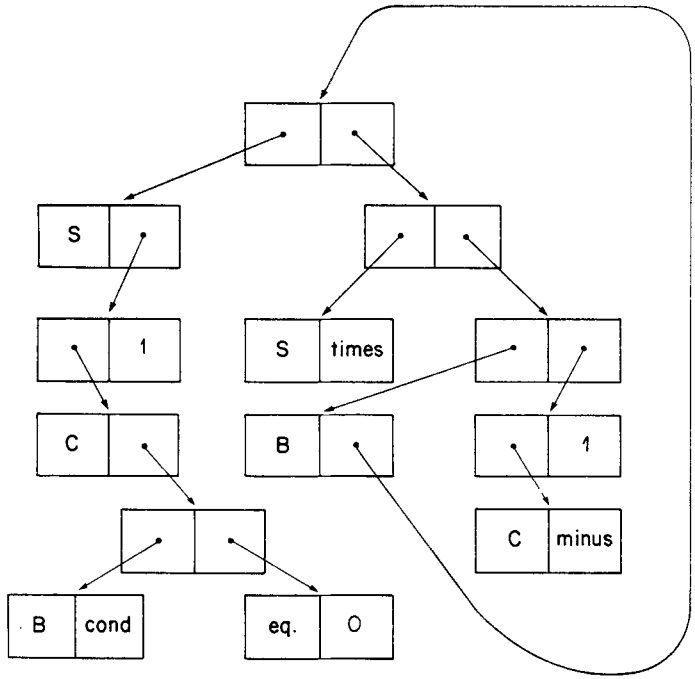

Figure 1. The code compiled for factorial

That is, for any function f, $\mathbf{Y}\,f$ is a fixed point of f. (See Burge[7] for further explanation of the use of **Y** in eliminating recursion.)

Mutual recursion following a **where** is handled by combining all the definitions following the **where** into a single definition with a complex left hand side and then proceeding as above. So for example

$$E_1 \quad \textbf{where}\ f\,x = \ldots g \ldots$$
$$g\,y = \ldots f \ldots$$

is first transformed to

$$E_1 \quad \textbf{where}\ f = [x]\ (\ldots g \ldots)$$
$$g = [y]\ (\ldots f \ldots)$$

eliminating the variables x and y. Now the mutually recursive pair of definitions can be converted into a single recursive definition as follows

$$E_1 \quad \textbf{where}\ (f,g) = ([x]\ (\ldots g \ldots), [y]\ (\ldots f \ldots))$$

which can be compiled as

$$([f, g]\ E_1)\ (\mathbf{Y}\ ([f,g]\ ([x]\ (\ldots g \ldots), [y]\ (\ldots f \ldots))))$$

using the rules already given.

In handling a moderately complicated definition then it is quite possible for the compiler to get a dozen or so levels deep in abstraction operations. In order that the code produced should continue to be reasonably compact under these circumstances it was found to be necessary to introduce some further optimizations into the abstraction algorithm. These take account of the way successive abstractions interact to bring about a considerable increase in the efficiency of nested abstraction. We omit the details here as they are fully described elsewhere.[8]

THE S–K REDUCTION MACHINE

Having described the action of the compiler we now pass to an account of the run time system. When the user submits to the SASL system an expression to be evaluated the compiler removes all the variables and passes to the run time system a binary graphical structure as described in the previous section. The run time system consists of a machine (currently implemented in software) which progressively transforms this structure by applying the following reduction rules until it has been reduced to a number or other printable object. (In these rules lower case letters stand for arbitrary structures unless otherwise specified.)

$$\mathbf{S}\,f\,g\,x \Rightarrow f\,x\,(g\,x)$$

$$\mathbf{K}\,x\,y \Rightarrow x$$

$$\mathbf{Y}\,h \Rightarrow \text{(see Figure 3)}$$

$$\mathbf{C}\,f\,g\,x \Rightarrow (f\,x)\,g$$

$$\mathbf{B}\,f\,g\,x \Rightarrow f\,(g\,x)$$

$$\mathbf{I}\,x \Rightarrow x$$

$$\mathbf{U}\,f\,(\mathbf{P}\,x\,y) \Rightarrow f\,x\,y$$

$$\mathbf{cond\ true}\,x\,y \Rightarrow x$$

$$\mathbf{cond\ false}\,x\,y \Rightarrow y$$

$$\mathbf{plus}\,m\,n \Rightarrow m+n$$ where m, n must already have been reduced to numbers.

And similarly for **times**, **minus**, **divide**, etc. The name of the machine is taken from the first two rules.

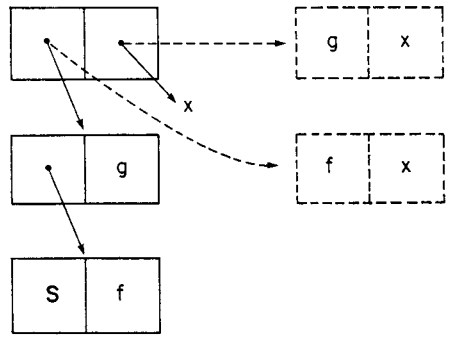

Figure 2. The effect of an **S***-reduction. The dotted lines show the state after the reduction has been performed*

It should be understood that these reduction rules are implemented as graph-transformation rules, that is to say that a node which matches the left hand side of a rule is overwritten with the corresponding right hand side. By way of illustration the effect of the rule for **S** is shown in Figure 2.

One rule in particular, that for **Y**, takes advantage of the graphical representation in an essential way. The reduction rule for **Y** given in the textbooks is

$$\mathbf{Y}\,h \Rightarrow h\,(\mathbf{Y}\,h)$$

Applying this rule again to the **Y** in the resulting expression we get $h\ (h\ (\mathbf{Y}\ h))$ and one can imagine obtaining after an infinite number of steps the infinite expression

$$h\ (h\ (h\ (\ldots)))$$

whose finite representation as a cyclic graph is shown in Figure 3. Our reduction rule for **Y** produces this form directly in one step, with a considerable gain in efficiency. (Note then that a recursive definition whether local or global always results in the construction of a cyclic graph.)

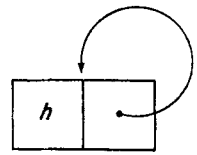

Figure 3. The result of reducing **Y** *h*

Before going on to discuss further details of the working of the machine it may be helpful to follow right through the processing of a simple example. Suppose the user types on his console as an expression to be evaluated

$$suc\ 2 \quad \textbf{where}\ suc\ x = 1 + x$$

First of all the compiler transforms this to

$$([suc]\ (suc\ 2))\ ([x]\ (1 + x))$$

thereby producing the code

$$\mathbf{C\ I}\ 2\ (\textbf{plus}\ 1)$$

The reduction machine progressively transforms this to

I (**plus** 1) 2	using the **C**-rule
plus 1 2	using the **I**-rule
3	using the **plus**-rule

and the number 3 is then printed as the system's response to the user's request. The form of the compiled code and the first step in its reduction are shown in Figure 4.

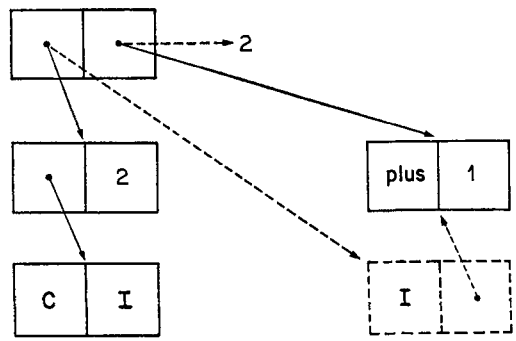

Figure 4. The code compiled for $suc\ 2$ **where** $suc\ x = 1 + x$ *and the first step in its reduction*

Quite apart from the use of combinators this machine is unconventional in that its operation consists in the progressive transformation of the compiled code. This is a very different mode of operation from that of a conventional 'fixed program' machine in which

the code is not normally altered once it has been compiled. In contrast we can call the kind of machine considered here a 'substitution machine' because it carries out literal substitutions on the compiled program. What is usually considered the principal objection to self-modifying code, namely that it is not re-entrant (i.e. cannot be shared between different uses) does not apply here because the transformations consist always in the replacement of an expression by another to which it is mathematically equivalent (for example, '**plus** 1 2' by '5').

Such a machine can be considered as a kind of *simplifier*, analogous to an algebraic simplifier. The program, in an applicative language, is an expression, which can be read as a mathematical description of the output the user desires to produce. The action of the machine consists in progressively simplifying this description, by applying rules known to preserve meaning, until it reaches a form from which the output can be printed directly.

Normal graph reduction

A major policy decision not yet touched on is the order in which the machine carries out the reductions, for in general more than one redex (redex = instance of the lhs of a rule) will be present at any given stage. We know on theoretical grounds that the final outcome of a reduction sequence is independent of the order in which the reduction rules are applied. One reduction sequence will differ from another, however, in the number of steps taken to reach the final outcome, including for certain initial expressions the possibility that one sequence will terminate while another fails to do so (for example, by getting stuck in a loop).

The ordering rule actually used is that at each stage we carry out reduction of the *leftmost* redex present. This is so called 'normal order reduction', which aside from being very simple and convenient to administer, as we shall see shortly, has the advantage that it is known to bring about termination whenever termination is possible. The other widely studied reduction regime is 'applicative order reduction' in which no redex is reduced until all redexes internal to it have been dealt with. To see that this terminates less often, suppose that a function which discards its argument and returns a constant answer (for example '**K** 2') is applied to a non-terminating sub-expression. Here evaluation will terminate under normal order but not under applicative order, where it will be attempted, in vain, to reduce the sub-expression completely before discarding it (i.e. normal order supports non-strict functions while applicative order does not).

Normal order reduction, it should be noted, means that in general an argument is substituted into the body of a function in unevaluated form (because initially the redexes inside the argument are to the right of other redexes). Mechanisms of this sort have a reputation for being inefficient on the grounds that they lead to repeated re-evaluation of the argument (cf Algol call by name). Here it is necessary to stress the importance of the fact that we are working with graphs. After the substitution all occurrences of the argument in the body of the function will be pointers to a shared sub-expression and all simultaneously 'feel the benefits' of any reductions carried out on the argument. So normal *graph* reduction, which is what we have here, combines the safety advantage of normal order (arguments are not evaluated needlessly) with the efficiency advantage of applicative order (arguments are evaluated at most once). See Wadsworth[9] for a fuller discussion of the properties of normal order reduction on graphs.

(It should be mentioned in passing that another possible reduction regime, with the same termination properties as normal order but a potentially enormous gain in speed, is at each stage to perform all available reductions *in parallel.* Foreseeable developments in

computer hardware might make this or some suitable variant an attractive possibility, but we shall not pursue this possibility here.)

Controlling the sequence of reductions

To schedule the sequence of reductions in normal order we use a 'left ancestors stack' which initially contains only (a pointer to) the expression to be evaluated. So long as the expression at the front of the stack is an application we keep taking its *left* subtree (the function of the function–argument pair) and pushing that onto the stack also. Eventually we shall get an atom at the front of the stack. If it is a combinator we apply the appropriate reduction rule, using the pointers in the stack to gain efficient access to its arguments (the *n*th argument of the combinator will be the right subtree of the object *n* places behind it on the stack). In Figure 5 we show the state of the stack before and after applying the **C**-rule

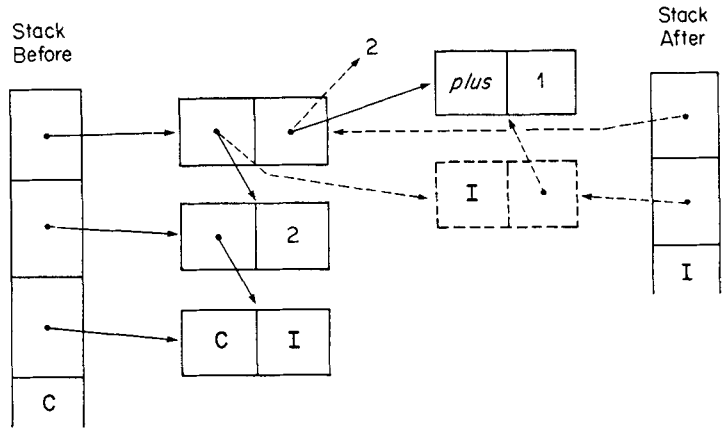

Figure 5. Behaviour of the stack during a reduction

of Figure 4 (stacks are drawn growing downwards). After applying a rule we resume stacking left components until we reach another atom. If the object at stack front is an arithmetic operator, say **plus**, we must call the reduction procedure recursively to reduce its arguments to numbers and then apply the appropriate rule. Note that the rules for **U** and **cond** also require reduction of one of their arguments before they can be used.

The sequence of reductions continues in this way, being controlled at each stage by the form of the expression at stack front, until eventually we have only one item left in the stack and that is a number or some other printable object. For brevity we have omitted from this account the possibility of an error arising such as that a combinator is found to have too few arguments or that an operator has arguments of the wrong type or that a non-function turns up as the leftmost item of a combination. Unless the compiler includes complete type-checking of the input text, which the author's currently does not, it is necessary to arrange to be able to detect such situations at run time and give appropriate error reports.

The user may submit for evaluation an expression whose value is a data structure (a list or a list of lists or whatever) instead of a number or a truthvalue or similar simple object. In this case the reduction algorithm will leave residing on the stack a representation of the data structure in which the components will not yet have been reduced. The printing routine prints the data structure from right to left, calling up the reduction algorithm to simplify each component before it is printed (and in case the component is itself a data

structure this process will be recursive). So the whole process of normal graph reduction can be thought of as being driven by the need to print—nothing is evaluated until the printer requires to know its value.

Among the intriguing possibilities to which this gives rise is that the user can submit an infinite list to be printed, for example the list of all prime numbers as defined by the SASL expression shown in Figure 6. For a fuller discussion of this and other properties of normal order implementations of applicative languages see Turner.[12]

```
sieve (from 2)
where
from n = n: from (n+1)
sieve (p : x) = p: sieve (filter x)
                where
                filter (n : x) =
                      n rem p = 0 → filter x;
                      n: filter x
```

Figure 6. The list of all the prime numbers

Storage allocation and indirection nodes

All the structures manipulated by the run-time system are built out of two-field cells which makes it convenient to use a LISP-style storage allocation scheme with markbits and a garbage collector.

Whenever a reduction rule is applied, it will be recalled, the node to which the rule is being applied is *overwritten* with the result. A problem arises if the result of applying the rule is either at atom or an already existing expression rather than a new node, as occurs for example when applying the rule for **K**. In the first case we have a problem because we have to overwrite a two-field cell with an object that can only occupy one field; in the second case we have a problem because if we overwrite the subject node with a copy of the top node of the result we shall destroy the sharing properties on which the efficiency of normal graph reduction depends.

The solution in both cases is to make the node into an *indirection node* with the identity combinator **I** in its left field and the result in question (an atom or a pointer to an existing expression) in its right field. On the stack of course we can leave the result itself rather than a pointer to the indirection node. Figure 7 shows by way of example the reduction of a redex of the form **K** x y.

In order to save space and time, pointers via indirection nodes are elided whenever they are encountered during processing. That is to say whenever we come across a field containing a pointer to a node of the form **I** x we overwrite the field with x. When all references to the indirection node have been bypassed in this way the space it occupies will of course be reclaimed by the garbage collector. It would also be possible to include a search for and elision of all pointers via indirection nodes during the mark phase of garbage collection.

As a final point on storage allocation it should be noted that the use of mark-scan garbage collection rather than a reference count technique is necessitated by the fact that the graphs which constitute the code are in general cyclic, because of the way we handle recursion. The occurrence of cycles could be avoided if all recursion (global as well as local) were done via the use of **Y** and if the reduction rule use for **Y** were

$$\mathbf{Y}\ h = h\ (\mathbf{Y}\ h)$$

rather than the 'knot-tying' rule shown in Figure 3. The graphs would then be and remain

acyclic, permitting the use of reference count techniques of storage allocation. This would be, however, at the cost of a considerable loss of efficiency, especially in the implementation of global recursions.

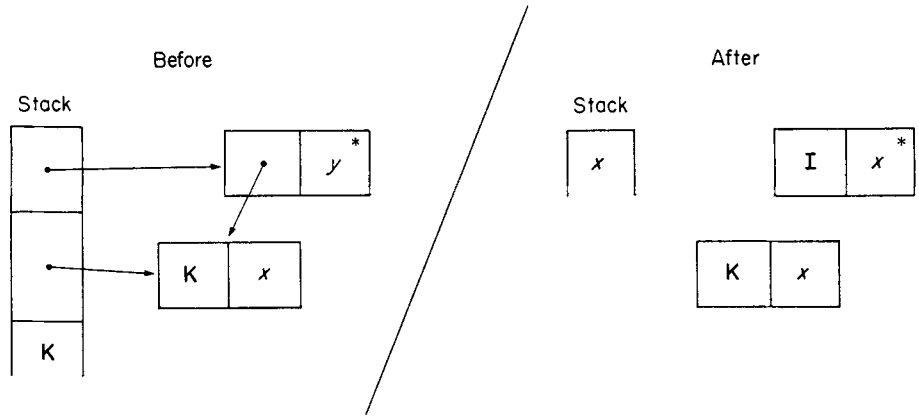

Figure 7. **K**-*reduction, note that the cell marked* * *becomes an indirection node*

RESULTS AND CONCLUSIONS

The compiler and the reduction machine described above have been implemented by the author and some measurements of performance taken, which we present in this section. An earlier implementation of the SASL language by means of a conventional interpreter using association lists was available for comparison.[13] The two implementations are written in different languages for different machines so there are a number of incidental differences between them. Any inferences from the relative performance of the implementations to the relative merits of the techniques are therefore tentative.

The first comparison made was in the compactness of the object code produced by the two compilers. The earlier implementation consists of an SECD machine together with a compiler for generating a suitable SECD machine-code. By way of illustration the code generated for the body of the factorial function (as defined in the introduction) is shown in Figure 8. Anyone familiar with SECD machines should find the code in Figure 8 self-explanatory—for details see Reference 13. Comparing this with the combinatory code shown for the same function in Figure 1 we see that the latter occupies 13 cells while the SECD code needs 24 cells, not counting the space occupied by the names '*n*' and '*fac*'.

This ratio seems fairly typical—the author finds that in general the combinatory form is about twice as compact as the SECD code, not counting the latter's extra storage requirement for bound variables. Over several hundred lines of source the SECD compiler averaged 14 cells of object code per line of source, the combinatory compiler six cells per line (not counting space occupied by names in either case). It should be stressed that the optimizations for multiple abstraction described in Reference 9 proved essential to maintaining this degree of compactness.

Execution speed

The second comparison made was in the relative speeds of execution of the two object codes. A direct measurement of elapsed times would have been meaningless because of incidental differences between the implementations that would have greatly affected the

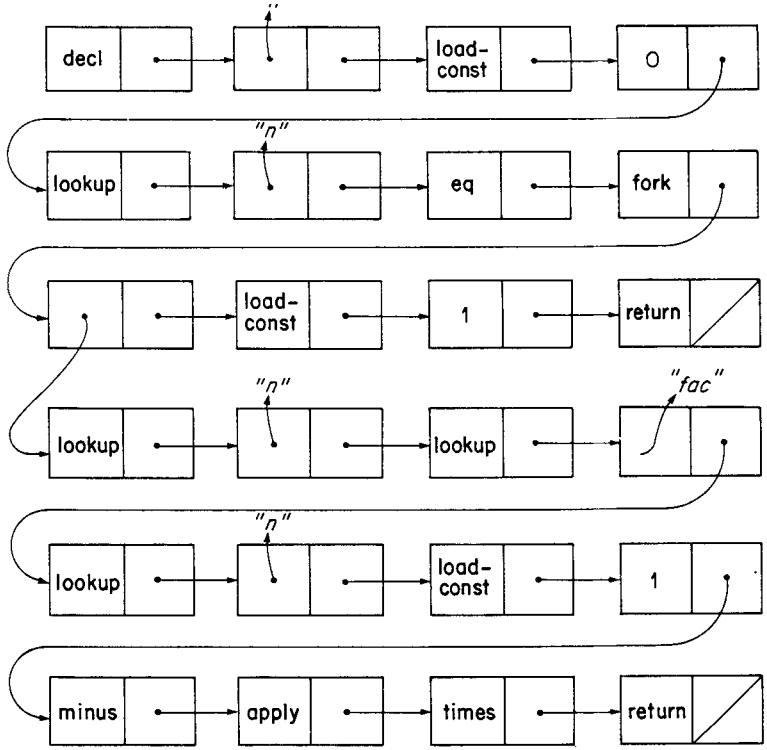

Figure 8. The SECD machine code for the body of factorial

(a)
```
hanoi 5 'a' 'b' 'c'
where
hanoi n a b c =
        n = 0 → nil;
        hanoi (n−1) a c b,
        "move a disc from", a, "to", b, "\n",
        hanoi (n−1) c b a
```

(b)
```
for 1 10 line
where
line n = "factorial", n, "is", fac n, "\n"
for a b f =
    a>b → nil;
    f a: for (a+1) b f
fac n =
    n = 0 → 1;
    n×fac (n−1)
```

(c)
```
twice twice twice suc 0
where
twice f x = f (f x)
suc x = x+1
```

Figure 9. Three test programs

timings, not least that they were running on different computers under different operating systems. One possible approximation to a machine independent measure of elapsed time is to count the number of 'steps' taken—i.e. SECD instructions in the one case and reductions performed in the other. Three sample programs are shown in Figure 9—(a) solves the

'Towers of Hanoi' problem with five disks, (b) prints a table of factorials and (c) is a calculation to test the use of higher order functions. Table I compares the number of steps in the execution of these test programs under the two implementations.

Table I. Number of steps

Program	SECD m/c	Reduction m/c
(a) Hanoi	1,488	3,067
(b) Factorials	975	1,280
(c) Twice	158	92

In general it seems to take the reduction machine rather more 'steps' to perform the same task, though this might be felt only to reflect the fact that the concept of a 'step' is here somewhat smaller.

Since both machines compose all their structures from two-field cells an arguably more comparable measure of 'work done' would be to count in each case the cumulative total of cells claimed from the store manager during execution (i.e. in LISP terms we are counting the total number of times CONS is called at run-time). The results for the same three test programs are shown in Table II.

Table II. Number of cells claimed during execution

Program	SECD m/c	Reduction m/c
(a) Hanoi	1,488	3,131
(b) Factorial	470	975
(c) Twice	128	65

In fact this measure can be seen to yield essentially the same results as those obtained by counting 'steps'. Notice that the relative cost of program (c), which is very dense in function calls, is much lower for the combinatory implementation—a point to which we shall return shortly.

We conclude then that the execution cost of programs is somewhat higher (perhaps by a factor of two) for the reduction machine than for the SECD machine (but for code dense in function calls the position is reversed).

A reservation must be made, however, on this comparison of execution costs. The reduction machine gives a *normal order* implementation, while the SECD machine gives an *applicative order* one. This means that the former implements a much more powerful version of the SASL language than the latter, for example in the ability to handle infinite data-structures. The SECD machine can be modified so that it also evaluates in (graphical) normal order. This is done by altering the parameter passing mechanism so that an actual parameter is passed as an unevaluated form (a 'suspension' containing pointers to the actual parameter expression and to the environment in which it is to be evaluated) and only later overwritten with its value if required.

Such a machine has been called 'procrastinating'[7] or 'lazy'.[10] We show in Table III the results of repeating the measurements of Table II on a 'lazy' version of the SECD machine coded by the author. We see that this modification slows the SECD machine down by *an*

order of magnitude. Compared with a 'lazy' fixed program machine then, the **S K**-reduction machine is much superior in execution speed.

Table III. Number of cells claimed during execution

Program	SECD m/c	LAZY SECD m/c
(a) Hanoi	1,488	10,428
(b) Factorial	470	5,565
(c) Twice	128	1,206

Self-optimizing properties

In contrast to programs in a conventional 'fixed program' machine, programs in the **S K**-reduction machine exhibit a number of important self-optimizing properties. First consider the case of a function whose body contains a sub-expression not involving any parameters or local variables of the function. During the first call of the function which requires this sub-expression to be evaluated it will be *replaced* by its value. That is to say the code for the function is permanently modified to that which would have been compiled if the user had written a constant in place of this sub-expression. As a consequence a constant calculation inside a loop is performed only once, regardless of how many times the body of the loop is executed.

A second aspect of this self-optimization concerns the cost of introducing extra levels of functional abstraction into a program. Take the following definition of a function to sum the elements of a list

$$\textbf{def}\ \mathit{sum}\ x =$$
$$x = \textbf{nil} \rightarrow 0;\ \textbf{hd}\ x + \mathit{sum}\ (\textbf{tl}\ x)$$

This represents a commonly occurring pattern of recursion which can be captured in the following definition of a generic function for 'folding a list to the right'

$$\textbf{def}\ \mathit{foldr}\ \mathit{op}\ a = f$$
$$\textbf{where}$$
$$f\ x =$$
$$x = \textbf{nil} \rightarrow a;$$
$$\mathit{op}\ (\textbf{hd}\ x)\ (f\ (\textbf{tl}\ x))$$

Now *sum* can be defined succinctly as

$$\textbf{def}\ \mathit{sum} = \mathit{foldr}\ \textbf{plus}\ 0$$

and many other functions now lend themselves to compact definitions in terms of *foldr*, for example

$$\textbf{def}\ \mathit{product} = \mathit{foldr}\ \textbf{times}\ 1$$
$$\textbf{def}\ \mathit{all} = \mathit{foldr}\ \textbf{and}\ \textbf{true}$$
$$\textbf{def}\ \mathit{some} = \mathit{foldr}\ \textbf{or}\ \textbf{false}$$

It will readily be appreciated that a systematic policy of defining and using generic functions like *foldr* whenever there is a commonly occurring pattern of recursion leads to

a programming style of great compactness and elegance—see for example Burge.[7] Such a 'combinatory' programming style involves of course a combinatorial increase in the number of function calls implied during the execution of the program. For a typical function instead of being defined directly is expressed as the application of a generic function to some previously defined functions, all of which may themselves have 'combinatory' definitions and so on to perhaps a considerable depth.

Under a conventional implementation the overhead of all the extra function calls places a definite limit on the extent to which such techniques can be exploited without intolerable slowdown. With the implementation technique described here, however, the overhead disappears entirely. To take the example in question, the first time *sum* is called the expression for sum, namely *foldr* **plus** 0 is replaced by the code associated with *f* in the body of *foldr*, with **plus** and 0 substituted for *op* and *a*. That is it is replaced by exactly the same code as would have been compiled from the direct definition of *sum*. So for the second and subsequent calls of *sum* the overhead associated with the programmer's introduction of the extra abstraction *foldr* simply disappears.

This property of the new implementation means that the programmer can freely introduce extra layers of abstraction whenever they contribute to modularity or to conciseness of expression without any loss in speed of execution. Such abstractions simply 'expand themselves out' the first time they are used.

Such behaviour is not peculiar to the **S K**-reduction machine—it would be shown by any *substitution* (as opposed to *fixed-program*) machine. For example, we could simply desugar our functional notation to λ-calculus at the compilation stage and have a machine which performed beta-reductions on the resulting λ-expressions (see Wadsworth[9] for a normal graph reducer for the λ-calculus). This would display the same self-optimizing behaviour. The advantage special to the use of combinators is that the associated reduction rules are much simpler and can be performed much faster than beta-reductions.

Some drawbacks

Under some circumstances the use of a substitution machine can have undesirable consequences for space utilization. It can be that the 'expanded out' form of an expression is so large that it would be preferable (or even essential) to lose it after each use and take the time to recreate it afresh from the original expression if it is needed again. (This is of course what always happens in a fixed-program machine.) It would be possible to give the programmer some special syntax for marking areas of the source text where this behaviour was wanted. Such a machine-oriented feature, however, would sit most oddly in an otherwise very clean high level language.

John Hammond, of the University of Kent, has suggested to the author that this problem could be solved by having the machine automatically discard expanded out versions in favour of original code whenever it runs out of space, rather after the manner of throw-away compiling.[14] Such a machine could display a continuum of possible behaviours ranging from full self-optimization to conventional fixed-program, depending on the amount of space available. Clearly further investigation of this matter is required.

The other main difficulty with the present implementation is that run-time error reports are very opaque. Descriptions of the run-time state in terms of the configuration of combinators on the stack are quite unintelligible to users. The system keeps a record, however, of the association of user-coined names with combinations of combinators as established by the user's **def** definitions. By using this 'dictionary' backwards, it should be possible to translate the information on the stack back into intelligible high level language expressions.

Here also further investigation is obviously needed. In fact the requirement for good run-time diagnostics will become less pressing in the future, since it is intended to introduce complete compile time type checking of the SASL source. Almost all programs that now cause run-time errors will then draw compile time diagnostics.

ACKNOWLEDGEMENTS

The author would like to thank Peter Welch and John Hammond of the University of Kent for many valuable suggestions made during frequent discussions during the course of the above work, Peter Collinson of the University of Kent for invaluable assistance in negotiating the interface with the host operating system, and Anthony Davie and Michael Weatherill of the University of St. Andrews for taking measurements on the earlier implementations of SASL.

REFERENCES

1. D. A. Turner, *SASL Language Manual*, University of St. Andrews, 1976.
2. P. J. Landin, 'The mechanical evaluation of expressions', *Comput. J.* **6**, 308 (1963–4).
3. J. McCarthy *et al.*, *LISP 1.5 Programmers Manual*, M.I.T. Press, 1962.
4. B. Randell and L. J. Russell, *The Implementation of Algol 60*, Academic Press, 1964.
5. M. Schonfinkel, 'Uber die Bausteine der mathematischen Logik', *Math. Annalen*, **92**, 305 (1924).
6. H. B. Curry and R. Feys, *Combinatory Logic*, Vol. I, North Holland, 1958.
7. W. H. Burge, *Recursive Programming Techniques*, Addison–Wesley, 1975.
8. D. A. Turner, 'Another algorithm for bracket abstraction', to appear in *Journal of Symbolic Logic*.
9. C. P. Wadsworth, *Semantics and Pragmatics of the λ-calculus*, Chapter 4, Oxford University; *DPhil. Thesis*, 1971.
10. P. Henderson and J. M. Morris, *A lazy evaluator*, 3rd Symposium on Principles of Programming Languages, Altanta, 1976.
11. D. P. Friedman and D. S. Wise, *CONS should not evaluate its arguments*, 3rd int. colloq. Automata Languages and Programming, Edinburgh, 1976.
12. D. A. Turner, *Programming without Assignment* (to appear).
13. D. A. Turner, *An Implementation of SASL*, University of St. Andrews, Dept. of Comp. Science, Report TR/75/4.
14. P. J. Brown, 'Throw-away compiling', *Software—Practice and Experience*, **6**, 423–434 (1976).