SETL Newsletter # 120

December 4,1973
J. Schwartz

A General-Recursive Extension of Functional Application, and its Uses.

The built-in operations of most languages are simply - recursive and thus allow only simply recursive operations to be written using built-in functions alone. Extra semantic power can be gained by making available a general-recursive primitive operation. It is especially appropriate to allow a binary operator op which accepts general list structures as its first parameter, in such a way as to make any general recursive function f(x) realisable as

represents_f op x.

Such an op will in effect be a general-purpose interpreter; its left-hand arguments representing programs in an internally manipulable form. A simple variant of this scheme is proposed in John Backus'two papers on 'reduction languages'; cf. Backus [1] and [2]. Note that a language with this one feature need in principle have no other features supporting control, recursion, or assignment; an observation which is however of more theoretical than practical interest. Such a primitive will of course make a variant of dynamic procedure formation available. The following SETL extension is freely adepted from Backus' proposal. We generalise the definition of the 'curly bracket' operation $f\{x_1,...,x_n\}$; this generalised 'application' becomes the op spoken of above. The following definition, which assumes a system mapping *defof* defined on blank atoms, is used.

$$
\begin{aligned}
f\{x_1,\ldots,x_n\} &= \langle x_1,\ldots,x_k\rangle \quad \text{if } f \text{ is the nullvector} \\
&= f(1)\ \{x_1,\ldots,x_n,\ f\} \text{ if } f \text{ is a vector;} \\
&= f(x_1,\ldots,x_{k-1},\ \langle x_k,\ldots,x_n\rangle) \text{ if } f \text{ is a function of} \\
&\qquad\qquad\qquad\qquad \text{k variables with } k<n \\
&= f(x_1,\ldots,x_n) \text{ if f is a function of a variable with} \\
&\qquad\qquad\qquad\qquad\qquad\qquad k = n \\
&= f(x_1,\ldots,x_n,\ \Omega,\ldots,\Omega) \text{ if } f \text{ is a function of k variables} \\
&\qquad\qquad \text{with } k>n \text{ and } x_n \text{ not a tuple} \\
&= f\{x_1,\ldots,x_{n-1},\ x_n(1),\ldots,x_n(\#\ x_n)\} \\
&\qquad \text{in all other cases if } f \text{ is a function} \\
&= f\{x_1,\ldots,x_n\} \text{ if } f \text{ is a set} \\
&= \text{defof}(f)\ \{x_1,\ldots,x_n\} \text{ if } f \text{ is blank} \# \ \Omega \\
&= \text{error in all other cases}
\end{aligned}
$$

It is clear that the generalised 'curly bracket application' described above can readily be programmed if the # operation can recover the number of parameters of a function and if an <u>apply</u> function is available for attaching argument list to objects. For this reason, we expand the list of SETL primitives very slightly, letting #f denote the number of arguments of f when f is a function, and introducing an opperator <u>apply</u> such that

(1) $$\underline{\text{apply}}\ \langle x_1,\ldots,x_n\rangle = x_1\ \{x_2,\ldots,x_n\}$$

Some examples: To construct an element f_1 such that

$$\langle f_1,f_2,\ldots,f_n\rangle\ \{x_1,\ldots,x_k$$
$$= f\ \{\langle f_2,\ldots,f_n\rangle\ \{x_1,\ldots,x_k\}\}$$

put $f_1 = \langle\ d,f\ \rangle$, where d is as below. Since

$$\langle f_1,\ldots,f_n\rangle\ \{x_1,\ldots,x_k\}$$
$$= \langle\ d,f\ \rangle\ \{x_1,\ldots,x_k,\ \langle f_1,\ldots,f_n\rangle\}$$
$$= d\ \{x_1,\ldots,x_k,\ \langle f_1,\ldots,f_n\rangle,\ \langle\ d,\ \ \rangle\}$$

we may define d by

```
definef d(u);
return u(#u)(2) {apply(<u(#u-1)(2:)> + u(1:#u-2))}
end d;
```

and then we have

$$\langle f_1,\ldots,f_n\rangle\ \{x_1,\ldots,x_k\} = f\ \{\langle f_2,\ldots,f_n\rangle\ \{x_1,\ldots,x_k\}\}.$$

This gives us a very easy 'composition' function

```
definef comp(f1,f2); return << d,f1 >,< d,f2>>; end comp;
```

To attach x as i-th parameter of an n-parameter function, getting an n-1 parameter function, use $\langle\ at,f,i,x\ \rangle$ with

```
definef at(u);
return u(#u)(2) { u(1:u(#u)(3)-1) + < u(#u)(4)> + u(u(#u)(3):)}.
end at;
```

To attach $g(x_1,\ldots,x_k)$ as i-th parameter of an n parameter function, getting an n+k-1 parameter function fg defined by $fg\ \{x_1,\ldots,x_{n+k-1}\} = f\ \{x_1,\ldots,x_{2-1},\ g\{x_1,\ldots,x_{1+k-1}\},x_{1+k},\ldots,\}$ we can proceed similarly using <subst,f,g,i,n>; where n may be Ω if f or g is a function. The program for *subst* is fairly obvious.

In situations where generalised application is to be used heavily, a syntax allowing applications with two arguments two be written in infix position is of course desirable.