On the power of list iteration

M. J. C. Gordon

Department of Computer Science, Edinburgh University, James Clerk Maxwell Building, The King’s Buildings, Mayfield Road, Edinburgh EH9 3JZ

The list iteration construct \( \text{lit} \) is defined by:

\[
\text{lit}([x_1, x_2, \ldots, x_n]; y; f) = f(x_1, f(x_2, \ldots, f(x_n, y) \ldots))
\]

Many simple list processing functions can be programmed as expressions built up from the LISP primitives (\textit{car}, \textit{cdr}, \textit{cons}, \ldots etc.) and \textit{lit}, examples include most of the operations of Boolean algebra and primitive recursive arithmetic (with the integer \( n \) represented by a list \((\text{NIL} \ldots \text{NIL})\) of \( n \) \text{NIL}'s). This latter example shows that the equivalence of list iteration programs is undecidable.

In the paper the use of \textit{lit} is illustrated and two theorems are proved which exhibit limitations on its power; these enable one to show that, for example, LISP's \textit{equal} and \textit{flatten} (where e.g. \textit{flatten} \([(1 2 3) 4 (5 6)]) = (1 2 3 4 5 6)) cannot be computed by list iteration alone.

(Received November 1977)

In statement oriented languages (e.g. ALGOL) the use of iteration constructs such as \textit{while}'s and \textit{for} loops is widely believed to be preferable to the use of unrestricted jumps. Similarly when using a functional language (e.g. LISP) there are advantages in avoiding unrestricted recursion. In this paper I examine the computational power of one well known form of restricted recursion—the list iteration construct \textit{lit} of Barron and Strachey (1966).

Suppose \( x = (x_1, x_2, \ldots, x_n) \) and \( y \) are LISP \( S \)-expressions* and \( f : S \times S \rightarrow S \) then define:

\[
\text{lit}(x; y; f) = f(x_1, f(x_2, \ldots, f(x_n, y) \ldots))
\]

\textit{lit} is a generalisation of the \( \Sigma \) and \( \Pi \) notation for iterated sums and products:

\[
\sum_{i=1}^{n} x_i = \text{lit}((x_1, \ldots, x_n); 0; \text{plus})
\]

\[
\prod_{i=1}^{n} x_i = \text{lit}((x_1, \ldots, x_n); 1; \text{times})
\]

Many simple list processing functions can be expressed using \textit{lit} for example the concatenating function \textit{append} is given by:

\[
\text{append}(x; y) = \text{lit}(x; y; \text{cons})
\]

to see this observe that:

\[
\text{append}((x_1, x_2, \ldots, x_n); (y_1, \ldots, y_m)) = \text{lit}((x_1, x_2, \ldots, x_n); (y_1, \ldots, y_m); \text{cons})
\]

\[
\text{cons}(x_1; \text{cons}(x_2; \ldots, \text{cons}(x_n; (y_1, \ldots, y_m)) \ldots))
\]

Similarly the reversing function \textit{rev} may be defined by:

\[
\text{rev}(x) = \text{lit}(x; \text{NIL}; \lambda([y_1; y_2]; \text{append}(y_2; \text{list}(y_1))))
\]

\[
\text{lit}(x; \text{NIL}; \lambda([y_1; y_2]; \text{lit}(y_2; \text{list}(y_1)); \text{cons}))
\]

Programs written in this way, as well as being more structured', can be efficiently implemented since \textit{lit} can be coded iteratively.

In what follows we examine what can and cannot be computed with expressions built up from the LISP primitives and \textit{lit}. First I will illustrate the kind of thing that can be done by describing some (mostly well known) examples. Next I will intuitively discuss two theorems which exhibit limitations of \textit{lit}—these allow one to show that certain functions cannot be computed with it alone. Finally I will rigorously define a notion of 'lit-computable' (which is designed to formalise the practice of using \textit{lit} as illustrated in the examples) and then prove the two theorems.

\*Throughout this paper I will be using the notation and terminology of LISP as described in the LISP 1.5 Programmer’s Manual (McCarthy et al, 1969). Also \( S \) and \( N \) will denote the sets of \( S \)-expressions and non-negative integers respectively.

Some examples of list iteration

To illustrate the power of \textit{lit} we describe how Boolean algebra and primitive recursive arithmetic (Rogers, 1967) can be programmed. We will represent a set by a list of its members (e.g. \([x_1, \ldots, x_n]\) is represented by \((x_1, \ldots, x_n)\)) and an integer \( n \) by a list of \( n \) \text{NIL}'s (e.g. \( 3 \) is represented by \((\text{NIL} \text{NIL} \text{NIL})\)).

First Boolean algebra:

1. If \( \text{member}(x; y) \) is \( T \) whenever \( x \) is a member of \((\text{the set represented by}) y \) and \( F \) otherwise, then:

\[
\text{member}(x; y) = \text{lit}(y; F; \lambda([z_1; z_2]; [eq(z_2; T) \rightarrow T; eq(z_1; x) \rightarrow T; T \rightarrow F]))
\]

e.g. \( \text{member}(2; (1 2 3)) = T; \text{member}(2; (1 4 3)) = F \)

2. If \( \text{union}(x; y) \) is a list representing the unions of \((\text{the sets represented by}) x \) and \( y \), then:

\[
\text{union}(x; y) = \text{lit}(x; y; \lambda([z_1; z_2]; \text{member}(z_1; y) \rightarrow z_1; \text{member}(z_2; y) \rightarrow z_2; T \rightarrow \text{cons}(z_1; z_2)))
\]

e.g. \( \text{union}((1 2 3); (2 4 5)) = (1 2 3 4 5) \)

3. If \( \text{intersection}(x; y) \) is a list representing the intersection of \( x \) and \( y \), then:

\[
\text{intersection}(x; y) = \text{lit}(x; y; \text{NIL}; \lambda([z_1; z_2]; \text{member}(z_1; y) \rightarrow \text{cons}(z_1; z_2); \text{member}(z_2; y) \rightarrow \text{cons}(z_1; z_2)))
\]

e.g. \( \text{intersection}(1 2 3; (1 3 4)) = (1 3) \)

4. If \( \text{complement}(x; y) \) is a list representing the complement of \( x \) in \( y \), then:

\[
\text{complement}(x; y) = \text{lit}(y; \text{NIL}; \lambda([z_1; z_2]; \text{member}(z_1; y) \rightarrow z_2; T \rightarrow \text{cons}(z_1; z_2)))
\]

e.g. \( \text{complement}(1 2 3; (1 2 3)) = (2) \)

5. If \( \text{boolean}(x) \) is a set representing the set of all subsets of \( x \), then:

\[
\text{boolean}(x) = \text{lit}(x; \text{list}(\text{NIL}); \lambda([y_1; y_2]; \text{lit}(y_2; y_2; \lambda([z_1; z_2]; \text{append}(z_2; \text{cons}(\text{cons}(y_1; y_2); \text{NIL}))))))
\]

e.g. \( \text{boolean}((1 2 3)) = (\text{NIL} (3) (2 3) (2 1 2) (1 2 3) (1 3) (1)) \)

6. If \( x_1, x_2, \ldots, x_n \) represent sets \( s_1, \ldots, s_n \) respectively and \( \text{cartprod}(x_1, \ldots, x_n) \) represents \( s_1 \times s_2 \times \ldots \times s_n \) then:

\[
\text{cartprod}(x) = \text{lit}(x; \text{list}(\text{NIL}); \lambda([y_1; y_2]; \text{lit}(y_2; y_2; \text{NIL}; \lambda([z_1; z_2]; \text{lit}(y_2; z_2; \lambda([w_1; w_2]; \text{cons}(\text{cons}(z_1; w_1); w_2)))))))
\]
I will now show how to simulate primitive recursive arithmetic; besides illustrating the (perhaps) surprising power of *lit* this also shows that the equivalence of the kind of programs described above is undecidable.

If the integer *n* is represented by a list \((NIL \ldots NIL)\) of *n* \(NIL\)’s then *n* is represented by \(NIL\), the (constant) zero function by \(\lambda[x]; NIL\) and the successor function by \(\lambda[x]; cons[NIL; x]\). Suppose \(f\) is defined (in terms of *g* and *h*) by the following primitive recursive definitions:

\[
\begin{align*}
&f(0; x_1; \ldots; x_n) = g[x_1; \ldots; x_n] \\
&f(\lambda; x_1; \ldots; x_n) = h[\lambda; f(y; x_1; \ldots; x_n); x_2; \ldots; x_n]
\end{align*}
\]

then I will show that:

\[
\begin{align*}
f(\lambda; x_1; \ldots; x_n) &= \text{lit}[\text{predecessors}][y]; g[x_1; \ldots; x_n]; \\
&\quad \lambda[x_1; z_1]; h[z_1; x_2; \ldots; x_n]]
\end{align*}
\]

to see this observe that:

\[
\begin{align*}
f(\lambda; x_1; \ldots; x_n) &= \begin{cases} 
\text{if } y = 0 & g[x_1; \ldots; x_n] \\
\text{if } y > 0 & h[y-1; f[y-1; x_1; \ldots; x_n]; x_2; \ldots; x_n]
\end{cases}
\end{align*}
\]

which ‘unwinds’ to

\[
\begin{align*}
f(\lambda; x_1; \ldots; x_n) &= h[y-1; h[y-2; \ldots; h[0; g[x_1; \ldots; x_n]; x_2; \ldots; x_n]; x_3; \ldots; x_n]; x_1; \ldots; x_n] \\
&= \text{lit}[y-1; 0]; g[x_1; \ldots; x_n]; \\
&\quad \lambda[x_1; z_1]; h[z_1; x_2; \ldots; x_n]; x_1; \ldots; x_n]
\end{align*}
\]

where \(\text{predecessors}\) can be defined with \(\text{lit}\) by:

\[
\begin{align*}
\text{predecessors}[y] &= \begin{cases} 
\text{NIL} & \text{if } y = 0 \\
\text{cons}[\text{cons}[\text{cons}[x_1; \text{car}[z_2]]; z_2]] & \text{if } y > 0
\end{cases}
\end{align*}
\]

Thus any primitive recursive function can be programmed as an expression built up from the LISP primitives and \(\text{lit}\). Such a definition uses no recursion other than that encapsulated in \(\text{lit}\) —i.e. no recursion other than that in the definition:

\[
\text{lit}[x; y; f] = \text{atom}[x] \rightarrow y; T \rightarrow f[\text{car}[x]; \text{lit}[\text{cdr}[x]; y; f]]
\]

**Limitations on the power of list iteration**

Having looked at what can be done with \(\text{lit}\) I will now discuss some results which provide insight into what cannot. I shall call a function ‘lit-computable’ if it can be programmed using \(\text{lit}\) in the way illustrated informally above (lit-computable is rigorously defined in the next section). For example the functions \(\text{append, rev, member, union, etc.}\) are all lit-computable. It follows from the theorems to be proved that, for example, \(\text{equal}\) (where \(\text{equal}[x; y] = T\) if \(x = y\) and \(\text{equal}[x; y] = F\) otherwise) is not lit-computable. The main idea in these theorems is that the maximum amount of work \(\text{lit}[x; y; f]\) can do is determined by the length of \(x\) —lit can only apply \(f\) at most \(\text{length}[x]\) times. In general, however, \(\text{lit}\) may be applied not just to \(x\) but also to sublists of \(x\). Thus the ‘energy’ of a list \(x\) is determined not simply by the (top level) length of \(x\) but the maximum of the length of all sublists of \(x\) —i.e. by \(\text{max} \text{lit}[x]\) where \(\text{max} \text{lit}\) is defined by:

**Theorem 1**

If \(f\) is lit-computable then:

\[
\forall m \exists n. \ (x_1 \times \ldots \times x_n) \leq m \Rightarrow \text{max} \text{lit}[f][x] \leq n
\]

From this it follows, for example, that if \(\text{flatten}\) is the function which removes all the inner brackets from a list, e.g.

\[
\text{flatten}((1 (2 3) (4 5) 6)) = (1 2 3 4 5 6)
\]

then \(\text{flatten}\) is not lit-computable. For suppose it were and consider \(\text{flatten}[x]\) where \(x_1 = 1, x_2 = ((1)), x_3 = ((1)(1))\) etc., then for any \(i > 1 \cdot \text{max} \text{lit}[\text{flatten}][x] = \text{max} \text{lit}[1][\ldots][1] = i\) but \(\text{max} \text{lit}[x] = 2\). Now taking \(m = 2\) in Theorem 1 there must exist an \(n\) such that:

\[
\forall i. i = \text{max} \text{lit}[\text{flatten}][x] \leq n
\]

but this is clearly impossible.

The second theorem concerns the ability of lit-computable functions to distinguish between similar lists. Let \(x = (\alpha) y\) mean that \(x\) and \(y\) are the same up to depth \(n\), i.e.

\[
x = (\alpha) y \Leftrightarrow x = y \lor (\text{atom}[x] = F \land \text{atom}[y] = F)
\]

and

\[
\text{car}[x] = (\alpha) \text{car}[y] \land \text{cdr}[x] = (\alpha) \text{cdr}[y]
\]

Then Theorem 2 shows that if \(\text{max} \text{lit}[x]\) and \(\text{max} \text{lit}[y]\) are bounded then every lit-computable function \(f\) has the property that \(f[x]\) can be made the same as \(f[y]\) up to a given depth by making \(x\) the same as \(y\) up to some depth.

**Theorem 2**

If \(f\) is lit-computable then:

\[
\forall m \exists n. \ (x_1 \times \ldots \times x_n) \leq m \land \text{max} \text{lit}[y] \leq m \land \text{max} \text{lit}[x] \leq m
\]

\[
x = (\alpha) y \Rightarrow f[x] = (\alpha) f[y]
\]

This theorem can be used to show equal is not lit-computable, for suppose it was then the function \(\text{equal}\) defined by \(\text{equal}[x] = \text{equal} \text{car}[x] \land \text{equal} \text{cdr}[x]\) would also be. Now consider \(\text{equal} \text{car}[x]\) and \(\text{equal} \text{car}[y]\) where:

\[
x_0 = (T. T) \quad y_0 = (T. F) \\
x_1 = ((T) . (T)) \quad y_1 = ((T) . (F)) \\
x_2 = (((T)) . (((F)))) \\
\ldots
\]

For all \(i \text{ max} \text{lit} [x_i] = \text{max} \text{lit} [y_i] = 2\) and \(x_i = (\alpha) y_i\) so taking \(m = 2\) and \(n = 0\) in Theorem 2 there exists a \(k\) such that:

\[
\forall i. x_i = (\alpha) y_i \Rightarrow \text{equal}[x_i] = (\alpha) \text{equal}[y_i]
\]

but this is impossible for we take \(i = k\) then this entails:

\[
T = \text{equal} \text{car}[x] = (\alpha) \text{equal} \text{car}[y] = F
\]

and it is not the case that \(T = (\alpha) F\).

**Formalisation and proof**

Before proving the theorems discussed above it is necessary to formalise lit-computability. I shall do this by defining a little expression language called LIT (a subset of LISP in fact) and then a function will be lit-computable if and only if there exists an expression to compute it. The proofs will then be structural inductions (Burrstall, 1969) over the set of all expressions.

Let \(I\) range over identifiers (strings of letters and digits beginning with a letter) and let \(A\) range over LISP atoms (we assume \(A\) contains \(NIL, T, F\) at least), then the syntax of LIT expressions \(E\) is specified by:

\[
E::= I | \text{car}[E] | \text{cdr}[E] | \text{atom}[E] | \text{cons}[E_1; E_2] | \text{eq}[E_1; E_2] | \text{lit}[E_1; \ldots; E_n] | \lambda[I_1; \ldots; I_n] ; E_1
\]

the subscripts on \(E\)’s are just to distinguish different occurrences. I shall denote the sets of expressions, identifiers and atoms by \(E\), \(I\) and \(A\) respectively. As expressions may contain free variables (i.e. identifiers) their meaning is only defined relative to an assignment of values (namely S-expres-
The semantics of LIT is specified by a function $\mathcal{E}: \text{Exp} \to (U \to S)$; the definition of $\mathcal{E}$—which is given below—should be read in conjunction with the explanatory notes which follow it.

**Semantics of LIT**

**Environments:** $U = \text{Ide} \to S$

**Semantic function:** $\mathcal{E}: \text{Exp} \to (U \to S)$

**Semantic equations:**

$\mathcal{E}[u] = u[I]$  
$\mathcal{E}[A]u = A$  
$\mathcal{E}[\text{car}(E)]u = \text{car}(\mathcal{E}[E]u)$  
$\mathcal{E}[\text{cdr}(E)]u = \text{cdr}(\mathcal{E}[E]u)$  
$\mathcal{E}[\text{atom}(E)]u = \text{atom}(\mathcal{E}[E]u)$  
$\mathcal{E}[\text{cons}(E_1, E_2)]u = \text{cons}(\mathcal{E}[E_1]u, \mathcal{E}[E_2]u)$  
$\mathcal{E}[\text{eq}(E_1, E_2)]u = \text{eq}(\mathcal{E}[E_1]u, \mathcal{E}[E_2]u)$  
$\mathcal{E}[\text{list}(E_1, \ldots, E_n)]u = \text{list}(\mathcal{E}[E_1]u, \ldots, \mathcal{E}[E_n]u)$  
$\mathcal{E}[\text{lit}(E_1 \to E_2, \ldots, E_{n+1} \to E_{n+2})]u = \text{lit}(\mathcal{E}[E_1]u, \mathcal{E}[E_2]u, \ldots, \mathcal{E}[E_n]u, \mathcal{E}[E_{n+1}]u)$  
$\mathcal{E}[\lambda x. E]u(x; I_1, I_2)]y(x; y)]$  

**(Notes)**

1. $\mathcal{E}$ is defined by structural induction on Exp; the emphatic brackets $[], []$ are just to aid the eye; they always enclose expressions of LIT.

2. I have systematically confused LISP atoms and functions with their meanings—i.e. I have used the same notation for both. Thus inside $[]$ NIL, $T$, $F$, car, cdr, atom, cons, eq, list, lit are just parts of expressions whilst elsewhere they denote the well known atoms and list processing functions, viz:

\begin{align*}
\text{NIL}, T, F & : S \\
\text{car}, \text{cdr}, \text{atom}: S \to S \\
\text{cons}, \text{eq} & : S^2 \to S \\
\text{list} & : S^* \to S \\
\text{lit} & : S \times S \times [S^2 \to S] \to S
\end{align*}

I shall assume that all these functions are total, e.g. that $\text{car}(\text{NIL})$ is defined (say to NIL), etc.; as will become apparent the theorems are not sensitive to fine details which I will therefore omit. To help reduce confusion between LIT and its metalanguage (e.g. between car as a symbol of LIT and as a function of $S$) I will surround arguments to functions in the latter with round brackets—thus $\text{car}(\text{NIL})$ in Exp, $\text{car}(\text{NIL})$ in $S$ etc.

3. If $u: U \to S$ and $I: \text{Ide}$ then $u[s/I]$ is the environment identical to $u$ except the $s$ is assigned to $I$, i.e.

$u[s/I] = \lambda I. I$  
$u[I_1] = u[I]$

I shall use the variables $E, I, A, u, s, m, n, k$ (possibly with subscripts) to range over particular sets as follows:

$E$ will range over the set $\text{Exp}$ of LIT expressions

$I, A, u, s, m, n, k$ (possibly with subscripts)

$U$ will range over the set $\text{Env}$ of environments

$s, m, n, k$ will range over $S$, the set of LISP S-expressions

$m, n, k$ will range over $N$, the set of non-negative integers.

I shall assume variables are initialised to NIL—i.e. the 'top-level' environment is $u_{init}$ defined by:

$\mathcal{E}[u_{init}] : U$ is defined by: $\forall u \in \mathcal{E}[u_{init}] = \text{NIL}$

lit-computability can now be rigorously defined.

**Definition 1**

$u_{init} : U$ is defined by: $\forall u \in \mathcal{E}[u_{init}] = \text{NIL}$

**Definition 2**

A function $F: S^* \to S$ is lit-computable if and only if:

$\exists E \forall s_1, \ldots, s_n \cdot F(s_1, \ldots, s_n) = \mathcal{E}[E]u_{init}[s_1/x_1, \ldots, s_n/x_n]$

**Remark:** The choice of identifiers $x_1, \ldots, x_n$ here is arbitrary—any $n$ distinct identifiers would do just as well.

**Definition 3**

If $u: U$ then $\max 1(u) = \sup \{ \max 1(u[I]) \mid I \in \text{Ide} \}$

**Lemma 1**

$\forall E, m \in \mathbb{N} \cdot (\forall u. \max 1(u) \leq m \Rightarrow \max 1(\mathcal{E}[E]u) \leq m)$

**Proof**

Induction on the structure of $E$:

If $E = I$ then $\max 1(u) \leq m \Rightarrow \max 1(u[I]) = \max 1(u[I]) \leq m$ by Definition 3.

If $E$ is an atom $A$ or atom ($E_1, E_2$) or eq($E_1, E_2$) then $\mathcal{E}[E]u$ is an atom so $\max 1(\mathcal{E}[E]u) = 0$ and the lemma is trivially true.

If $s_1, s_2, \ldots, s_n \in S$ then:

$\max 1(\text{car}(s_i)) \leq \max 1(s_i)$

$\max 1(\text{cdr}(s_i)) \leq \max 1(s_i)$

$\max 1(\text{cons}(s_i, s_j)) \leq \max 1(s_i, s_j)$

$\max 1(\text{list}(s_i, \ldots, s_j)) \leq \max 1(s_i, \ldots, s_j)$

and so if the lemma holds for $E_1, E_2, \ldots, E_n$ then it holds also for $\text{car}(E_i), \text{cdr}(E_i), \text{cons}(E_i, E_j)$ and $\text{list}(E_i, \ldots, E_n)$.

max($\mathcal{E}[E_1]u, \ldots, \mathcal{E}[E_{n+1}]u, \mathcal{E}[E_{n+2}]u) \leq \max(\max(\mathcal{E}[E_{n+1}]u, \ldots, \mathcal{E}[E_{n+2}]u)$

then if the lemma holds for $E_1, E_2, \ldots, E_{n+2}$ then it holds also for $\text{lit}(E_1 \to E_2, \ldots, E_{n+1} \to E_{n+2})$.

**Theorem 1**

If $F: S \to S$ is lit-computable then:

$\forall m \exists n \forall s. \max 1(s) \leq m \Rightarrow \max 1(F(s)) \leq n$

**Proof**

If $F$ is lit-computable then there exists $E$ such that:

$\forall s. F(s) = \mathcal{E}[E]u_{init}[s/x_1]$

by lemma 1:

$\forall m \exists n \forall u. \max 1(u) \leq m \Rightarrow \max 1(\mathcal{E}[E]u) \leq n$

now $\max 1(u_{init}) = 0$ so $\max 1(u_{init}[s/x_1]) = \max 1(s)$. Hence

$\forall m \exists n \max 1(s) \leq m \Rightarrow \max 1(\mathcal{E}[E]u_{init}[s/x_1]) \leq n$

Theorem 1 follows, QED.
\textbf{Definition 4}
If \( u_1, u_2; U \) then \( u_1 = (u_2) u_2 \Leftrightarrow \forall I. u_1[I] = (u_2) u_2[I] \).
In what follows I will use the obvious facts that
\( u_1 = u_2 \Leftrightarrow \forall n. u_1 = (u_2) u_2 \) and
\( n \geq m \Rightarrow (u_1 = (u_2) u_2 \Rightarrow u_1 = (u_2) u_2) \).

\textbf{Lemma 2}
\[ \forall E, m, n \exists k. (u_1, u_2 \cdot \text{max} (u_1) \leq m \text{ and} \text{max} (u_2) \leq m \Rightarrow \text{max}(u_1 u_2) \leq m) \]
\[ (u_1) \text{max} (u_1) \leq m \Rightarrow (u \Rightarrow u_1 \Rightarrow \text{max}(E) u \Rightarrow \text{max}(E) u_1) \]

\textbf{Proof}
Structural induction on \( E \):
If \( E = I \) then (by Definition 4) we can take \( k = n \)
If \( E \) is an atom then Lemma 2 is clearly true.
It follows from the meaning of the LISP primitives that for all \( k \):
\( s = (s_1 + s_1) \Rightarrow \text{car}(s) = (s_1 \text{car}(s_1)) \)
\( s = (s_1 + s_1) \Rightarrow \text{cdr}(s) = (s_1 \text{cdr}(s_1)) \)
\( s = (s_1 s_1) \Rightarrow \text{atom}(s) = (s_1 \text{atom}(s_1)) \)
\( s_1 = (s_1 s_1) \Rightarrow s_2 = (s_1 s_2) \Rightarrow \text{cons}(s_1 s_2) = (s_1 \text{cons}(s_1 s_2)) \)
\( s_1 = (s_1 s_1) \Rightarrow s_2 = (s_1 s_2) \Rightarrow \text{equiv}(s_1 s_2) = (s_1 \text{equiv}(s_1 s_2)) \)
\( s_1 = (s_1 s_1) \Rightarrow s_2 = (s_1 s_2) \Rightarrow \text{list}(s_1, s_2) = (s_1 \text{list}(s_1, s_2)) \)

hence if Lemma 2 holds for \( E_1, E_2, \ldots, E_n \) then it also holds for \( \text{car}(E_1), \text{cdr}(E_1), \text{atom}(E_1), \text{cons}(E_1, E_2), \text{equiv}(E_1, E_2) \) and \( \text{list}(E_1, \ldots, E_n) \).

If \( \delta[E_1, u_1] = (u) \delta[E_1, u_1] \text{ and } \delta[E_2, u_2] = (u) \delta[E_2, u_2] \) then \( \delta[E_1, u_2] = (u) \delta[E_1, u_2] \text{ and } \delta[E_2, u_2] = (u) \delta[E_2, u_2] \) and hence if Lemma 2 holds for \( E_1, E_2, \ldots, E_n \) then it also holds for \( \delta[E_1, \ldots, E_n] \).

Finally if \( E = \text{lit}(E_1, E_2, \ldots, E_{n+1}, E_{n+2}) \) and \( n \) and \( n+1 \) are given then by Lemma 1 there exists \( k_0 \) such that:
\[ \max(u_1) \leq m \Leftrightarrow \text{length}(\text{max}(E) u_1) \leq \max(\text{max}(E) u_1) \leq k_0 \]
and so by induction on \( E_1 \) there exists \( k_1 \geq k_0 \) such that:
\[ \max(u_1) \leq m, \max(u_1) \leq m \Rightarrow (u = (u_1) u_1 \Rightarrow \text{max}(E_1) u_1 = (u) \text{max}(E_1) u_1) \]

and hence if \( E_{0}, E_1, E_2 \) are as in the last part of the proof of Lemma 1 then:
\[ \max(u_1) \leq m, \max(u_1) \leq m \Rightarrow (u = (u_1) u_1 \Rightarrow \text{max}(E_1) u_1 = (u) \text{max}(E_1) u_1) \]
and so if we choose, by induction, a \( k_1 \geq k_0 \) such that for all \( i \leq k_0 \)
\[ \max(u_1) \leq m, \max(u_1) \leq m \Rightarrow (u = (u_1) u_1 \Rightarrow \text{max}(E_1) u_1 = (u) \text{max}(E_1) u_1) \]

then it follows that:
\[ \max(u_1) \leq m, \max(u_1) \leq m \Rightarrow (u = (u_1) u_1 \Rightarrow \text{max}(E_1) u_1 = (u) \text{max}(E_1) u_1) \]

\textbf{QED.}

\textbf{Theorem 2}
If \( F : S \to S \) is lit-computable then:
\[ \forall m, n \exists k \forall s, s'. \max(s) \leq m \text{ and} \max(s') \leq m \Rightarrow (u = (u_1) u_1 \Rightarrow F(s) = (u) F(s')) \]

\textbf{Proof}
If \( F \) is lit-computable then there exists \( S \) such that:
\[ \forall s . F(s) = \text{max}(E)[u_1 \text{lit}(s)] \]

by Lemma 2:

\textbf{References}

The Computer Journal Volume 22 Number 4

\textbf{Acknowledgements}
I learnt about \textit{lit} from Rod Burstall who supervised the research reported here. The definitions of \textit{cartprod} and \textit{boolean} are due to C. Strachey and D. Du Feu respectively, the simulation of primitive recursive arithmetic is due to G. Plotkin. Finally I would like to thank John Allen, Peter Buneman, Rod Burstall, John Darlingon, Gordon Plotkin and Jim Thatcher, all of whom made helpful suggestions or criticisms at various stages of the work.