Extended Abstract

Field Lucid - Programming with Time and Space
by
A. Faustini

Declarative Languages and Advanced Architecture Laboratory
Department of Computer Science
Arizona State University
Tempe
USA

Introduction

Lucid is an established programming language in which programs are written using an implicit notion of time. In this paper we present an augmentation of Lucid that permits programs to be written using an implicit notion of space. The extended language is called Field Lucid and it properly includes the standard Lucid family of Languages.

Ashcroft and Wadge have proposed an extension of Lucid to deal with space as well as time. Their proposal involved the introduction into the language of a number of spatial operators. These operators were first introduced by Ashcroft and Wadge in the Lucid book and were called initial, rest and cby.

In the first implementation of Field Lucid a new set of spatial operators was introduced, different from those proposed by Wadge and Ashcroft. These new operators were generalizations of first, next and fby into higher dimensions. In fact it was a misunderstanding of the original operators that produced the alternate spatial operators.

A paper by Ashcroft entitled "Ferds- Massive Parallelism in Lucid" introduced a further set of spatial operations that where a generalization to multi-dimensional space of the operators first described in the Lucid book.

This paper we will examine the merits of these different spatial operations. The question that we try to answer is which of these families of operators are useful. We will try to do this through the use of the various operators in different programming examples.

The Original Book Lucid Operators
The following are the original definitions of the spatial operations first given in the Lucid Book

\[(A \ cby \ B)_t s_0, s_1, \ldots = \begin{cases} \begin{align*} A_t s_1, s_2, \ldots & \quad \text{if } s_0 = 0 \\ B_t s_0, s_1, \ldots & \quad \text{otherwise} \end{align*} \end{cases}\]

\[(restA)_t s_0, s_1, s_2, \ldots = A_t s_0+1, s_1, \ldots \]

\[(initialA)_t s_0, s_1, s_2, \ldots = A_t 0, s_0, s_1, \ldots\]

Operators from the first implementation

The following are the definitions used in the first Field Lucid implementation. We shall these operators names that are suggestive of the fact that they are generalizations of first, next and fby. Note that in this case the k subscript is a positive integer the zero component being reserved for the usual time dimension t :

\[(A \ fby k B)_t s_1, s_2, \ldots = \begin{cases} \begin{align*} A_t s_1, \ldots, s(k-1), s(k+1), \ldots & \quad \text{if } s_k = 0 \\ B_t s_1, \ldots, s(k-1), s(k), s(k+1), \ldots & \quad \text{otherwise} \end{align*} \end{cases}\]

\[(next k A)_t s_1, s_2, \ldots = A_t s_1, \ldots, s(k-1), s(k+1), s(k+1), \ldots\]

\[(first k A)_t s_1, s_2, \ldots = A_t s_1, \ldots, s(k-1), 0, s(k+1), \ldots\]

Ferds paper operators

\[(A \ cby k B)_t s_0, s_1, s_2, \ldots = \begin{cases} \begin{align*} A_t s_1, \ldots, s(k-1), s(k+1), \ldots & \quad \text{if } s_k = 0 \\ B_t s_1, \ldots, s(k-1), s(k), s(k+1), \ldots & \quad \text{otherwise} \end{align*} \end{cases}\]

72
\[
(restk A)_{t} s_{0}, s_{1}, s_{2}, \ldots = A_{t} s_{0}, s_{1}, \ldots, s_{(k-1)}, s_{k+1}, s_{(k+1)}, \ldots
\]
\[
(initialk A)_{t} s_{0}, s_{1}, s_{2}, \ldots = A_{t} s_{0}, s_{1}, \ldots, s_{(k-1)}, 0, s_{(k)}, \ldots
\]

Example Programs

The following is the outline of a program to solve Laplace's equation in a two dimensional space of size \(N\) by \(M\), using the usual relaxation method where the next value at each point in space is the average of its four neighbors.

\[
s \text{ asa settled}\n\]
\[
\text{where}\n\]
\[
s = \text{ORIG fby cond}\n\]
\[
\text{OUTSIDE} : 0; \n\]
\[
\text{ELECTRODE} : \text{POTENTIAL}; \n\]
\[
\text{default} : f(s) ; \n\]
\[
\text{end}; \n\]
\[
f(A) = (\text{up } A + \text{ down } A + \text{ left } A + \text{ right } A) / 4; \n\]
\[
\text{ORIG} = \text{if ELECTRODE then POTENTIAL else 0 fi}; \n\]
\[
\text{OUTSIDE} = \text{x\_coord} = \text{N} \text{ or x\_coord} < \text{0} \text{ or y\_coord} = \text{M} \text{ or y\_coord} < \text{0}; \n\]
\[
\text{end}
\]

Note that \(n\_\text{coord} = 0\ cbyi n\_\text{coord} + 1\), like index, is usually a built-in feature of the language. In addition to operators like \text{next} and \text{rest}k\ that enable the programmer to access the next time or space point; there are also special operators that enable the programmer to access the previous time or space point. The reader should be able to contract their definitions from the above definitions of \text{next} and \text{rest}k.\ What happens when we apply a "previous" operator at the origin of time or space? We go into negative time or negative space. To facilitate this we extend the time and space indices to permit negative indices. In the above program operators like \text{up} and \text{right} are really \text{rest}0\ and \text{rest}1\ respectively. The \text{down} and \text{left} operators are the "previous" space point operators for space dimension 0 and 1 respectively.
The following extended example of the Laplace program for 3 dimensions is a good example of how easy it is to extend programs to higher dimensions:

```plaintext
s asa settled
where
  s = ORIG fby cond
      OUTSIDE : 0;
      ELECTRODE : POTENTIAL;
      default : f(s);
  end;
  f(A) = (up A + down A + left A + right A + front A + rear A) / 6;
  ORIG = if ELECTRODE then POTENTIAL else 0 fi;
  OUTSIDE = x_coord=N or x_coord<=0 or y_coord=M or
             y_coord<0 or O=y_coord or z_coord<0;
end
```

We present two versions of the sieve of Eratosthenes one written with time and function calling and the other written in time and space. In the first program a new recursive call is made of sieve each time a new prime is to be output. Thus the number of primes output is directly proportional to the depth of the function calling. Each call of sieve can be thought of as spawning a new coroutine. The first call is a coroutine that removes multiples of the first element of nats (i.e. 2) from the stream nats. The 2 is output as a prime and nats stream with multiples of 2 removed is sent to a new call of sieve which does exactly the same as the first in that it removes all multiples of the first element 3 of the new input stream and so on. The reader should spend some time to try to understand what is going on in this program as it is very instructive. Note that the activations of sieve never die off; they simply provide additional filtered numbers to deeper calls.

```plaintext
sieve(nats) where
  nats = 2 fby nats+1;
  sieve(n) = n fby sieve( n wvr n mod first n ne 0);
```

74
end;

The following is the time/space version of the same program. The primes in this program also appear in time; that is, the $i$th prime is the time $i$ (space 0) output of the program. An infinite vector of natural numbers beginning with $2, 3, 4, 5, \ldots$ is the starting point for an iterative process that removes all multiples of the first entry in the vector to produce the next vector in time:

```plaintext
initial sieve
  where
    sieve = nats fby sieve whvr sieve mod initial sieve ne 0;
    nats = 2 cby nats+1;
end;
```

Note that \texttt{whvr} is the space analog of \texttt{wvr}; in fact the definition of \texttt{whvr} is the same as \texttt{wvr} except that \texttt{fby} and \texttt{next} are replaced with \texttt{cby0} and \texttt{rest0}. We usually drop the subscripting when we are dealing with the first space dimension.

As a final challenge we ask the reader to explain why the following program produces as output a "running histogram" (in the first space dimension) of the distribution of natural numbers occurring in the time sequence \texttt{data} which is only partially defined in the program below. Any sequence of integers could be used and so we could substitute the rhs of the definition of \texttt{data} used below with \texttt{index mod 256}. In practice it would be an input to the module and so no definition would be required:

```plaintext
next histogram
  where
    histogram = index upon data eq x_coord;
    data = 2 fby 5 fby 67 fby 999 fby \ldots;  
end;
```

References
[Ash85] Ashcroft, E.A.,  
*Ferds - Massive Parallelism in Lucid*  

[Ash84] Ashcroft, E.A.,  
*The Syntax and Semantics of Lucid*  

*Lucid - A Formal System for Writing and Proving Programs.*  

*Lucid, a Nonprocedural Language with Iteration.*  

[Ash74] E.A. Ashcroft,  
*Towards an APL Compiler*  

[WaAs85] Wadge, W.W., and Ashcroft, E.A.  
*Lucid, the Dataflow Programming Language*  