A DATAFLOW LANGUAGE COMPILATION FOR A DEMAND-DRIVEN MACHINE

Rajiv Bagai
Department of Computer Science, Wichita State University, Wichita, KS 67260-0083, U.S.A.

Abstract. A declarative dataflow programming language, called Lucid, was invented by Wadge and Ashcroft over a decade ago. Here we describe the main features of Lucid with some example programs, and present a method for compiling Lucid source programs into equivalent C programs. The generated code emulates an abstract demand-driven machine. While multiprocessor architecture is ideal for Lucid, our compilation method makes it possible to execute Lucid programs on widely available uniprocessor systems.

Keywords. Dataflow programming, Demand-driven computation, Language translation.

1 Introduction

A promising alternative to the 'storage location' and 'command' concepts of conventional von Neumann architecture is the processing of data while it is 'in motion'. Commonly known as the dataflow model of computation, its fundamental characteristic is a network of asynchronously operating processing stations connected by channels through which data objects flow. While significant work has been done on the dataflow architecture and machines based on it (see, for example, [8, 7, 2, 12]), Wadge and Ashcroft [3, 4, 11] invented a programming language particularly suitable for such a computation model.

Lucid is a declarative programming language designed for the dataflow architecture. The language is based on intensional logic [10], which can manipulate expressions whose meaning depend on an implicit context. In Lucid this context is time, and so all its constants, variables and other expressions thereof denote time-varying streams of data objects.

While a multiprocessor architecture is ideal for implementing Lucid, since single-processor systems are still predominant, some work has been done on implementing Lucid upon them. Various concepts and strategies arising from these works were put into practice in the Lucid evaluator developed by Faustini [9]. In this evaluator Lucid is compiled into code for an abstract stack-based machine, which is simulated by its interpreter. Due to this interpreting mechanism and its extensive run-time type checking, the evaluator is very slow.

In contrast, this paper describes a method for compiling Lucid into machine executable code. A compiler based on this method was developed and tested for various Lucid source programs. Currently, the generated code is in the C language, and it emulates an abstract demand-driven evaluating machine.

The remaining part of this section gives a brief overview of Lucid. For a detailed exposition the reader is referred to [11]. Section 2 outlines the architecture of the underlying abstract machine used for evaluating Lucid programs. Section 3 describes the program translation process from Lucid into the C language. And finally, Section 4 contains some concluding remarks and directions for future work.
The Lucid language

A Lucid program is an expression denoting a time-varying stream of data objects, such as integers, booleans, character strings, etc. The time dimension is considered to be a sequence of points in time, and so a stream contains the values of an expression at times 0, 1, 2, ... For example, the Lucid program

\[
x + c
\]
\[
\text{where}
\]
\[
c = 2;
\]
\[
\text{end;}
\]

will input the values of the stream \( x \) at times 0, 1, 2, ... etc., and output a stream each of whose values is 2 more than the corresponding \( x \) value.

Every Lucid program represents a dataflow network of processing nodes connected by data channels. For example, the above program represents the simple network shown in Figure 1. Observe that the channels

\[
\begin{array}{c}
x \\
+ \\
2
\end{array}
\]

Figure 1: A simple dataflow network

in the network are capable of carrying infinite streams of data and the nodes are essentially operations over such infinite streams. (Constants are considered as nullary operations.)

In the above program, the stream \( x \) is pointwise added to the stream constant \( c \), which is defined to be the infinite stream \( \{2, 2, 2, \ldots\} \) by the where clause. Note that the Lucid constant \( 2 \) denotes the infinite stream \( \{2, 2, 2, \ldots\} \) instead of the scalar value 2.

The values of \( x \) are read as input because it is a free variable in the above expression. For example, if \( x \) is the infinite stream \( \{3, -1, 0, 11, \ldots\} \), then the whole program denotes the stream \( \{5, 1, 2, 13, \ldots\} \).

The '+' is an infix operator for pointwise addition of streams of numbers, i.e. the value of any expression \( x+y \) at any time \( t \) is the sum of the values of streams \( x \) and \( y \) at the same time. Notationally,

\[
(x + y)_t = x_t + y_t
\]

Other primitive pointwise operators, like unary '-' }, '*', div, mod etc. are defined similarly. And so are the relational, logical, string and list operators. The value of the conditional expression

\[
\text{if } p \text{ then } x \text{ else } y \text{ fi}
\]

at any time \( t \) is \( x_t \) if \( p_t \) is true, and \( y_t \) if \( p_t \) is false. Note that the types of the values \( x_t \) and \( y_t \) need not be the same.
The intensional operators

The pointwise operators introduced so far have a common property that the value of an expression involving those operators at time $t$ depends entirely upon the values of the operands at time $t$. In addition, Lucid provides intensional operators that make it possible to look back into the past or even ahead into the future of the values of the operands in arriving at the expression value.

The simplest of these is the unary operator $\text{first}$, which results in the constant stream obtained by propagating the time 0 value of its operand. It is defined as:

$$(\text{first } x)_t = x_0$$

For example, $\text{first}$ of the stream $(10, 20, 30, \ldots)$ is the stream $(10, 10, 10, \ldots)$.

The unary operator $\text{next}$ allows looking one step ahead in the future. It is defined as:

$$(\text{next } x)_t = x_{t+1}$$

i.e. the value of the stream $\text{next } x$ at any point in time is the value of the stream $x$ at the next point in time. So $\text{next}$ of the stream $(10, 20, 30, \ldots)$ is the stream $(20, 30, 40, \ldots)$.

The $\text{fby}$ (pronounced followed by) is a binary operator that, if possible, looks one step into the past of the values of its second operand. It is defined as:

$$(a \text{ fby } b)_t = \begin{cases} a_0 & \text{if } t = 0 \\ b_{t-1} & \text{if } t > 0 \end{cases}$$

which means that the value of the stream $a \text{ fby } b$ at time 0 is the value of the stream $a$ at time 0, and at any other point in time is the value of the stream $b$ at the previous point in time. For example, if $a$ is the stream $(10, 20, 30, \ldots)$ and $b$ is the stream $(100, 200, 300, \ldots)$, then $a \text{ fby } b$ is the stream $(10, 100, 200, 300, \ldots)$.

The most common usage of $\text{fby}$ is to define a stream recursively as in

$$\text{index} = 0 \text{ fby } (1 + \text{index})$$

This definition of the stream $\text{index}$, like all Lucid definitions, is an equation; the solution of this equation is the stream denoted by $\text{index}$. It is a stream whose time 0 value is 0, and whose value at any other point in time is the value of the expression $(1 + \text{index})$ at the previous point in time. In other words, $\text{index}$ denotes the stream $(0, 1, 2, \ldots)$. The following is a formal derivation of the same:

$$\text{index}_t = (0 \text{ fby } (1 + \text{index}))_t$$

$$= \begin{cases} 0 & \text{if } t = 0 \\ (1 + \text{index})_{t-1} & \text{if } t > 0 \end{cases}$$

$$= \begin{cases} 0 & \text{if } t = 0 \\ 1_{t-1} + \text{index}_{t-1} & \text{if } t > 0 \end{cases}$$

$$= \begin{cases} 0 & \text{if } t = 0 \\ 1 + \text{index}_{t-1} & \text{if } t > 0 \end{cases}$$

$0$ and $1$ are constant streams

The $\text{fby}$ operator is right associative; thus the expression $1 \text{ fby } 2 \text{ fby } 3$ denotes the stream $(1, 2, 3, 3, \ldots)$. Also, $\text{fby}$ is independent of the time $t > 0$ values of its first operand. A similar derivation as above will show that the program
fib
  where
  fib = 1 fby 1 fby (fib + next fib);
end;

denotes the Fibonacci sequence \(1, 1, 2, 3, 5, 8, \ldots\).

The operators \texttt{wvr} and \texttt{upon}

Two interesting primitive operators of Lucid are \texttt{wvr} (pronounced \textit{whenever}) and \texttt{upon}. Unlike the operators described so far, the values at time \(t\) of the expressions involving these two operators need not necessarily depend upon values at or around time \(t\) of their operands.

The binary operator \texttt{wvr} results in a stream made of those elements of its first operand for which the corresponding element of the second operand is \textit{true}. The first operand can be any stream, but the second should be a stream of boolean values. For example, if \(a\) is the stream \(0, 1, 2, \ldots\) and \(b\) is the stream \(\langle\text{false}, \text{true}, \text{true}, \text{false}, \text{false}, \text{true}, \ldots\rangle\), then a \texttt{wvr} \(b\) denotes the stream \(1, 2, 4, 7, \ldots\). It can therefore be thought of as a \textit{filtering} operator.

The binary operator \texttt{upon} has the effect of ‘slowing down’ its first operand. It results in a stream in which the elements of the first operand are repeated until a \textit{true} is encountered in the second operand. Like \texttt{wvr}, the first operand can be any stream, but the second should be a stream of boolean values. For example, for the same \(a\) and \(b\) as above, the expression \(a \texttt{upon} b\) denotes the stream \(0, 0, 1, 2, 2, 3, 3, 3, \ldots\).

A more practical example of the usage of \texttt{upon} is the following program:

\[
\begin{align*}
\text{if } xx <= yy \text{ then } xx \text{ else } yy \text{ fi}
\text{ where}
xx &= x \text{ upon } xx <= yy; \\
yy &= y \text{ upon } yy <= xx;
\end{align*}
\]

which denotes the stream obtained by merging the two non-descending input streams \(x\) and \(y\), with common values appearing only once. For example, if \(x\) is the stream \(0, 1, 1, 5, 10, \ldots\), and \(y\) is \(1, 4, 5, 5, 11, \ldots\), then \(xx\) is \(0, 1, 1, 5, 5, 10, 10, \ldots\), and \(yy\) is \(1, 1, 4, 4, 5, 5, 11, \ldots\). The stream denoted by the entire program is therefore \(0, 1, 1, 4, 5, 5, 10, 10, \ldots\).

This concludes our brief overview of the essential features of Lucid. The language actually has many other features and extended versions [11]. The author is currently exploring the incorporation of such dataflow features into declarative object-oriented settings, such as in [6, 1].

\section{The abstract machine}

The Lucid programs can be executed on a very simple abstract machine based on the lazy-evaluation strategy. Unless specified otherwise, the main objective of the machine is to output all values of the stream denoted by the source program. The order and the number of times the necessary intermediate expressions are evaluated may vary from one implementation to another, but their final output should be the same.
To get a better understanding of the behavior of the machine, consider the following program:

```
next s/n
   where
       s = 0 fby s + x;
       n = 0 fby n + 1;
   end;
```

which outputs the running average of the elements of the input stream \( x \). The default output of the machine for this program would be the values of the stream \( \text{next } s/n \) at times 0 onward. A demand for the value of the expression \( \text{next } s/n \) at time 0 will, due to the semantics of \( \text{next} \), generate a demand for the value of \( s/n \) at time 1. The '/' operator being pointwise, will in turn generate demands for \( s_1 \) and \( n_1 \). A demand for \( s_1 \), due to the definition of \( s_1 \), will reduce to a demand for \( (s + x)_0 \), which will generate demands for both \( s_0 \) and \( x_0 \). \( s_0 \) evaluates to 0, and \( x_0 \) is obtained by reading the first input value. So, \( s_1 \) evaluates to that first input value. Similarly, \( n_1 \) evaluates to 1. The value \( s_1/n_1 \) is therefore the same as \( x_0 \) and hence, so is \( (\text{next } s/n)_0 \). Similarly, \( (\text{next } s/n)_1 \) will evaluate to \( (x_0 + x_1)/2 \) and in general \( (\text{next } s/n)_t \) will be \( (\sum_{i=0}^{t} x_i)/(t + 1) \), which is what the program is supposed to compute.

Two observations are in order.

Firstly, even though the value of the stream \( s/n \) at time 0 is undefined, the stream denoted by the whole program is completely defined. The machine never attempts to evaluate \( (s/n)_0 \) since its value is not needed in order to produce the desired output. A data-driven technique on the other hand, in its eagerness to compute as much as possible, runs the risk of engaging one of its processors indefinitely on an undefined computation, or even if it is robust, may generate a 'divide-by-zero' error message for a program that does not deserve it.

Secondly, a demand for the value of a variable at a particular time coordinate generates demands for those variables that it depends upon, at possibly other coordinates. The demands grow in a tree-like fashion. All the internal nodes of this underlying demand tree correspond to the bound variables of the program. The leaf nodes correspond to Lucid constants and the free variables of the program.

The demands terminate at the constant nodes because they can propagate no further via that path and that Lucid constant evaluates to the scalar constant it represents.

The demands also terminate at the input variables because those variables are not defined in terms of others. The first time the value of an input variable \( v \) at time \( t \) is demanded, it is read from the associated input device, and possibly stored in some buffer for future references. All subsequent demands for \( v_t \) can just pick the value from the buffer. The need and sophistication of the buffering mechanism depend entirely upon the nature of the input device associated with \( v \).

Any such buffering mechanism modifies the underlying demand tree to a directed acyclic graph. Similar buffering can also be employed for the internal nodes of the demand tree to avoid recomputations. An efficient mechanism based on compile-time analysis of the source program appears in [5].

3 The compilation strategy

In this section we describe a two-step procedure for compiling a given Lucid program into an equivalent C program. The first step converts the Lucid program into an intermediate set of ordered pairs of variable definitions, and the second step converts this set into a C program that behaves like the original Lucid program.
Flattening a Lucid program

The main objective of this first compilation step is to flatten any multiple where clauses occurring in the Lucid program into an equivalent single where clause. Any where clause is essentially a set of variable definitions, and for the Lucid features presented in the paper so far the resulting where clause is very similar to the original program. It is obtained by performing a series of simple syntactic transformations [11] on the Lucid program. These transformations are aimed at 'simplifying' the original program while preserving its 'meaning'.

Given a Lucid program \( \rho \), its conversion is a logical derivation, i.e. a sequence of equivalent programs \( \rho_0 = \rho; \rho_1; \ldots; \rho_{n-1} \), where any \( \rho_{i+1} \) is an Lucid program obtained from \( \rho_i \) by one of the transformation rules. The derivation stops at \( \rho_{n-1} \), which is in the desired form and can be trivially translated to the target intermediate set of definitions.

Any Lucid expression can either be a simple-expression or a where-expression. A simple-expression is an expression involving any of the Lucid operators, and does not have a where clause. All variable occurrences in a simple-expression are therefore free. A where-expression is of the form

\[
\sigma \text{ where } \delta_0; \delta_1; \ldots; \delta_{m-1}; \text{end}
\]

where \( \sigma \), called the subject, can be any Lucid expression, and each \( \delta_i \) is a definition of the form

\[
\delta_i = \epsilon_i
\]

and binds the variable \( \theta_i \) with the Lucid expression \( \epsilon_i \). The underlying set of definitions \( \{\delta_0, \delta_1, \ldots, \delta_{m-1}\} \) is called compatible if no two definitions in the set bind the same variable. All variable occurrences in the subject of the where-expression that are defined by the definitions of its where clause are bound. All other occurrences are free, i.e. they have an 'outer' meaning.

Note that a simple-expression \( \sigma \) is semantically equivalent to the where-expression \( \sigma \text{ where end} \), which has an empty where clause. Therefore, without loss of generality, the term expression in the following discussion refers only to where-expressions (with a possibly empty set of definitions).

The renaming rule

The renaming rule states that the bound variables of any Lucid expression can be renamed as long as the renaming is safe.

A renaming is essentially any function over the set of all Lucid variables. For example, the renaming

\[
\begin{align*}
a & \rightarrow b \\
x & \rightarrow y
\end{align*}
\]

will result in substituting all occurrences of variable name \( a \) by \( b \), and \( x \) by \( y \) in any Lucid expression \( \epsilon \) that it acts upon. Moreover, the renaming is safe for \( \epsilon \) if the substitution process does not result in changing any free occurrence of a variable in \( \epsilon \) to a bound occurrence, and the compatibility of the underlying set of definitions is preserved. For example, the above renaming is unsafe for the expression

\[
a + b \\
\text{where } a = x + 2; \ x = 6; \ \text{end;}
\]

because the free variable \( b \) becomes bound. It is also unsafe for the expression

1114
\[ a + b \]
where \( a = x + 2; \ b = 6; \) end;

because the set of definitions in the where clause becomes incompatible.

**The concatenation rule**

The concatenation rule states that if the subject \( \sigma \) of a where-expression

\[ \sigma \text{ where } \delta_0; \delta_1; \ldots; \delta_{n-1}; \text{ end} \]

is itself of the form

\[ \varepsilon \text{ where } \xi_0; \xi_1; \ldots; \xi_{m-1}; \text{ end} \]

such that the set of definitions \( \{ \delta_0, \delta_1, \ldots, \delta_{n-1}, \xi_0, \xi_1, \ldots, \xi_{m-1} \} \) is compatible, then the whole expression can be transformed to

\[ \varepsilon \text{ where } \delta_0; \delta_1; \ldots; \delta_{n-1}; \xi_0; \xi_1; \ldots; \xi_{m-1}; \text{ end} \]

If the union of the two underlying sets of definitions is not compatible, then an appropriate renaming transformation of the subject \( \sigma \) should pave the way for the concatenation transformation of the entire expression.

**The amalgamation rule**

The amalgamation rule states that if \( \Theta(\varepsilon_0, \varepsilon_1, \ldots, \varepsilon_{n-1}) \) is a Lucid expression where each \( \varepsilon_i \) is an expression with subject \( \sigma_i \) and the underlying (possibly empty) set of definitions \( \Delta_i \), such that the union of all \( \Delta_i \)'s is compatible, then the entire expression can be transformed to

\[ \Theta(\sigma_0, \sigma_1, \ldots, \sigma_{n-1}) \text{ where } \bigcup_{i=0}^{n-1} \Delta_i \text{ end} \]

If the union is not compatible, then an amalgamation can be performed after an appropriate renaming transformation. For example, the Lucid expression

\[
\left( \begin{array}{c}
\text{a + b} \\
\text{where} \\
a = 1; \\
b = 2; \\
\text{end}
\end{array} \right) - \left( \begin{array}{c}
2 \times c \\
\text{where} \\
c = 3; \\
\text{end}
\end{array} \right)
\]

can be transformed under this rule to

\[(a + b) - (2 \times c) \]
where a = 1; b = 2; c = 3; end

**The liquidation rule**

This rule permits liquidation of a nested where clause into the enclosing one, if such a liquidation process is safe. Given an expression of the form

\[ \sigma \text{ where } \delta_0; \delta_1; \ldots; \delta_{n-1}; \text{ end} \]
with a definition \( \delta_k \) of the form

\[
\vartheta_k = \varepsilon_k \text{ where } \xi_0; \xi_1; \cdots; \xi_{m-1}; \text{ end}
\]

the expression can be transformed to

\[
\sigma \text{ where } \delta_0; \delta_1; \cdots; \delta_{k-1}; \vartheta_k = \varepsilon_k; \delta_{k+1}; \cdots; \delta_{n-1}; \xi_0; \xi_1; \cdots; \xi_{m-1}; \text{ end}
\]

if the underlying set of definitions of the resulting expression is compatible. As usual, if the set is not compatible, then an appropriate renaming transformation has to be performed before the liquidation rule can be applied.

**The final transformation**

Repeated applications of the transformation rules described above convert the Lucid source program into an equivalent Lucid program containing one (possibly empty) \texttt{where} clause. From this program the intermediate set of definitions can be obtained in a straightforward way. This set is essentially the underlying set of definitions of the \texttt{where} clause along with a special ordered pair \((\$\), \(\sigma\)), where \(\sigma\) is the subject of the \texttt{where} clause. For example, the Lucid program

\[
\begin{align*}
\texttt{a + b} \\
\texttt{where} \\
\texttt{a = c \text{ where } c = d + 1 \text{ end;}} \\
\texttt{d = 5;} \\
\texttt{end;}
\end{align*}
\]

is finally converted to the intermediate set of definitions \{ \(\{\$\), \(\{a, b\}\), \(\{c, d + 1\}\), \(\{d, 5\}\) \}.

**Creating the target C program**

The second and final step in compiling the given Lucid source program is to convert the intermediate set of definitions into a C program that behaves like the source program.

The generated C program is a self-contained emulation of the abstract machine described in Section 2, in that it may be compiled and executed independently to produce the desired behavior. The simplest organization of the program is to have two functions, namely \texttt{main()} and \texttt{VarEval()}. The function \texttt{VarEval()} evaluates the given variable at the given time coordinate. Thus the function \texttt{main()} simply invokes \texttt{VarEval()} for the imaginary variable \(\$\) at user-specified coordinates (default time coordinates 0, 1, 2, \ldots) and outputs the obtained values.

The \texttt{VarEval()} function is defined as

\[
\texttt{int VarEval(int v, int t)}
\]

and it returns the value of the variable \(v\) at time coordinate \(t\). In the implemented version of the compiler the returned values are restricted to be integers but in principle they can be of any type supported by Lucid. The variables are implemented as small integer values such as \(\_a\), \(\_b\), etc.. The \texttt{VarEval()} function is implemented as one \texttt{switch} statement with a case for each Lucid program variable (including \(\$\)). More precisely, there is one case in the \texttt{switch} statement for each tuple in the intermediate set obtained after the first compilation step. For example, for the tuple \(\{a, x + y - 2\}\) the corresponding C case would be
case _a:
    return VarEval(_x, t) + VarEval(_y, t) - 2;

As an example of the generated code, consider the running-average program introduced in the previous section:

next s/n
  where
    s = 0 fby s + x;
    n = 0 fby n + 1;
  end;

The VarEval() function for this program would be:

int VarEval(int v, int t) {
  int val;
  switch (v) {
    case _s: return VarEval(_s, t+1) / VarEval(_n, t+1);
    case _s: return t == 0 ?
               0 : VarEval(_s, t-1) + VarEval(_x, t-1);
    case _n: return t == 0 ?
               0 : VarEval(_n, t-1) + 1;
    case _x: printf("Please enter x["d]: ", t);
             fflush(stdout);
             scanf("%d", &val);
             return val;
  }
}

The first case is for the special variable $s$, which is defined by the subject of the where clause. It is easily understood by the following derivation for the value of $s_t$:

\[
    s_t = (next s/n)_t = (s/n)_{t+1} \quad \text{semantics of next}
    \]

\[
    = s_{t+1} / n_{t+1} \quad \text{semantics of } / 
\]

The cases for the variables $s$ and $n$ are the most illustrative, but since they are similar, we will walk through only the former one. The code for $s_t$ follows immediately from its semantic derivation:

\[
    s_t = (0 \ fby (s + x))_t = \begin{cases} 
        0 & \text{if } f = 0 \\
        s_{t-1} + x_{t-1} & \text{if } f > 0 
    \end{cases} \quad \text{semantics of } fby
\]

The variable $x$ occurs free in the Lucid program. Therefore, whenever a request for ‘computing’ its value at any given time coordinate arrives, the required value is obtained from the input device after giving an appropriate message. The buffering mechanism mentioned at the end of Section 2 can be employed to avoid having to reobtain the same value from an input device.

4 Conclusions

The Lucid language is particularly suitable for representing dataflow networks. While a multiprocessor architecture is ideal for implementing Lucid, we have presented a method to compile Lucid programs into
equivalent C programs for uniprocessor architecture.

The subset of Lucid considered in this paper was intentionally kept simple to enable easy explanation of the translation aspects. However, these methods can easily be extended to handle multi-dimensional time coordinates and other coordinates, such as space. We omitted discussing user defined functions because they involve elaborate extensions to the underlying compilation and abstract machine execution strategies.

The C program generated by the translation procedure makes heavy use of the \texttt{VarEval()} function, and consequently spends a significant portion of its execution time calling and returning from it. Context saving and restoration, and the passing of coordinates as parameters make the calls extremely expensive.

A better technique would be to generate code which, unlike the current one, is ‘sequential’ rather than ‘functional’ in nature. Also, implementing coordinates as global variables makes more efficient code possible. This is so because coordinates are usually propagated unchanged. An example \texttt{VarEval()} entry according to this strategy for the definition

\[ x = (a \times b) + c \times \text{next next d} - \text{next next e}; \]

would be:

```c
    case _x:
        if (t == 0)
            temp1 = VarEval(_a);
        else
            { t--; temp1 = VarEval(_b); t++; }
        t *= 2;
        temp2 = VarEval(_d); temp3 = VarEval(_e);
        t -= 2;
        return temp1 + VarEval(_c) * temp2 - temp3;
```

The variable \( t \) is a global time coordinate, and \texttt{VarEval()} just takes the variable as parameter. Such a code, generated manually by the author for some example programs, exhibited about 25% faster response than the one generated automatically by the compiler described in this paper.

References


1118


