

# The Lucid Language: Past, Present, and Future

*Edward Ashcroft*

*Computer Science and Engineering Department*

*Arizona State University*

*Tempe, Arizona 85287*

## *Abstract*

A description is given of the development of Lucid from its beginnings in 1974 to the present, together with a hint of possible things to come. While not a tutorial, the paper does convey most of the ideas behind Lucid.

## *Introduction*

Lucid began (in 1974) as an attempt by me and Bill Wadge to make iterative programs mathematically respectable without resorting to recursive functions or other semantic technical baggage. (We felt that recursion is too powerful a technique to be used for something as simple and intuitive as iteration.) In the process, we found ourselves with a language for which correctness proofs for programs could be both simple and natural [5, 6, 9, 10].

For a while, these two properties of Lucid were its only reasons for existence. An implementation was developed by a self-motivating undergraduate student at the University of Waterloo, Calvin Ostrum, who based the implementation on the mathematical semantics, and, in the process, invented a model of computation that we now call *eduction*. Using the implementation, when we started looking seriously at programs that weren't just Lucid versions of procedural programs - programs that were meaningful mathematically but whose operational meanings at first seemed obscure or "messy and weird" - we began to see that Lucid, the language, could be important in its own right. Since then all the work on Lucid has tended to concentrate on Lucid as a programming language - adding new features and capabilities [7], extending the implementation, reassessing and reformulating the programming paradigm (which we now see as *intensional programming*), finding new applications [3], designing an appropriate multiprocessor architecture, expressing Lucid programs visually, and maximizing

the implicit parallelism - rather than concentrate on Lucid as a new way of doing program-proving. Nevertheless, in all those 18 years, the mathematical semantics of Lucid has had primacy, and no features have been added to the language that would, for example, destroy referential transparency. This has had important consequences. For example, programs remain implicitly parallel and we have found that eductive implementations of Lucid are implicitly fault tolerant, both of which are properties that can be dramatically observed in GLU, a practical application of "Lucid technology." Moreover, the program-proving potential of Lucid is still intact, and still beckons some of us, siren-like.

This paper will start by outlining some of the basic features of Lucid from long ago, both as a programming language and as a program-proving paradigm. It will then describe the current situation and also indicate the various directions in which Lucid is moving while remaining firmly grounded in intensionality.

To a large extent, this discussion will be limited to work done at Waterloo, SRI, and ASU that I was aware of or participated in.

### *Early Days*

Originally, we wanted to make assignment mathematically respectable by defining a language in which assignment was equality.

One of the earliest Lucid programs was

```
first x = 1;
first y = 0;
result = y as _soon_ as x > z;
next x = x + 2*y + 3;
next y = y + 1;
```

This program was intended to mimic the FORTRAN subroutine body

```
      X=1.0
      Y=0.0
10  IF(X.GT.Z)GOTO20
      X=X+2.0*Y+3.0
      Y=Y+1.0
      GOTO10
20  PRINT,Y
      RETURN
```



The idea in Lucid was to mimic FORTRAN while avoiding statements like

**Y=Y+1.0**

which are nonsense mathematically, and sets of statements like

**X=1.0**

**Y=0.0**

**X=2.0\*Y+3.0**

which, taken together, are inconsistent. We wanted, as in FORTRAN, to have = as the assignment operator but we wanted it to really mean equality.

This was achieved by considering variables to denote sequences of values, i.e., the *histories* of the variables, and the operators **first** and **next** to be rather like Lisp's **car** and **cdr**. Arithmetic operators like + apply to sequences but work *pointwise*: the *i*-th value of **A + B**, for example, is the sum of the *i*-th value of **A** and the *i*-th value of **B**. (The same is true for the operator **if then else**.)

With this interpretation, in the above Lucid program **y** denotes 0, 1, 2, 3, .., and **x** denotes 1, 4, 9, 16, .. . We can *prove* that  $x = (y + 1)^2$ , by *Lucid Induction*: Clearly **first** **x** = (**first** **y** + 1)<sup>2</sup> (since **first** **x** is 1 and **first** **y** is 0), and if we assume that  $x = (y + 1)^2$  it is immediate that **next** **x** = (**next** **y** + 1)<sup>2</sup> by using a little high-school algebra (and the facts that **next** **x** is **x** + 2\***y** + 3 and **x** is (y + 1)<sup>2</sup>).

This proof uses the same high-school algebra as is used in a conventional verification of the corresponding FORTRAN program, using  $X = (Y + 1.0)^2$  as an invariant, but in Lucid there is no worry about where in the program the invariant has to be attached. In fact, in Lucid there is no concept of where the computation has reached; there is no flow of control. We can even *permute* the statements in the Lucid program and it means the same thing, and the proof is identical. (In the FORTRAN program, or any procedural program, the order of the updates is crucial.)

In Lucid, the statements in a program are essentially axioms from which properties of the program can be proved.

(The above Lucid program computes the integer square root of **z** but we will not complete the proof here.)

*Demand-Driven Computation*

In the above program, though it wasn't stated it should have been clear that the sequences denoted by  $x$  and  $y$  are *infinite*. To implement such a language, the elements in the sequences should only be calculated on demand. The sequences themselves can not be demanded, only the elements. The sequences are identified by the names of the variables they are the denotations of, and the elements in a sequence are all identified by their positions. For this early Lucid, a natural number (a position) will be considered to be a *context*, and the values of expressions are always demanded in particular contexts. When the expression is a variable, if the value of the variable in that context has already been calculated and stored then that stored value is the demanded value. If the value has not been stored then the demanded value will be the value produced by demanding, in the same context, the expression that is the right-hand-side of the definition of the variable. (When this value is produced it should be stored. The stored values will be associated with the appropriate variable and context, forming an associative memory.)

This context-dependent demand-driven evaluation technique is now called *eduction*. The first use of eduction in a Lucid implementation was made by Calvin Ostrum, in 1978. (Tom Cargill, a graduate student at Waterloo produced the *first* implementation of Lucid, written in ALGOLW, in 1976 or 1977. It was demand-driven but it did not store anything - it just re-evaluated when necessary. This may have been the first implementation of *any* language that used "lazy evaluation.")

### *User-Defined Functions*

Function-calling in Lucid obeys the "copy rule." If a function **square** is defined as follows

**square(n) = n\*n;**

then the value of **square(E)** in a given context is the value of  $E * E$  in the same context, that is, it is the value, in that context, of the expression defining **square**, with the formal parameter **n** replaced by the actual parameter **E**. (There are some complications due to the need to avoid clashes of local variables.) This explanation in terms of textual substitution does not convey the implementation technique actually employed. (If **E** were a very complicated or very hard to evaluate expression, we would not want to do the substitution, or do the evaluation, twice.) Essentially, we just want to evaluate the expression  $n * n$  but somehow take account of the fact that **n** means **E**. In conventional languages that is taken care of by "activation records," but that is avoided in Ostrum's eduction by just saying that we evaluate  $n * n$  in a context that indicates what is the actual parameter that corresponds to the formal parameter **n**. This means that contexts now are more complicated than a single natural number: they must also indi-



cate the "places" where functions have been called (because those places are where the actual parameters can be found). This information is easily incorporated into the contexts and associated with any values that are stored.

The result is that in an eductive implementation of Lucid there is no centralized "environment" to handle function-calling.

Originally, the copy rule was introduced as a way of defining the meaning of call-by-name in Algol. Unsurprisingly, therefore, the parameter-passing mechanism in Lucid is call-by-name also. The inductive implementation automatically results in call-by-name (or, rather, call-by-need).

It is easy to see that call-by-value will *not* work: Consider the truth-valued function **is-increasing** defined by

**is-increasing(a) = a < next a;**

which is *not* pointwise. Suppose we had to evaluate **is-increasing(E)** for some expression **E** in some context *i*. We cannot use call-by-value. We cannot evaluate **E** in context *i* and then pass that value as the argument when we call **is-increasing**. The function needs the values of **E** both in context *i* and in context *i* + 1, and that can not be determined before the function is called (without some analysis of the definition of the function, and such analysis is exactly the sort of extra computation that call-by-value is supposed to avoid). Lazy evaluation is needed.

In summary, function calling fits easily into the eductive evaluation mechanism and automatically gives lazy evaluation of functions.

### *Syntactic Improvements*

We felt that the syntax of Lucid could be improved. With the existence in Lucid of the ability to define nonpointwise functions, it was clear that Lucid was not destined to just mimic FORTRAN programs. We therefore were not concerned when, after we changed the syntax, programs no longer directly corresponded to FORTRAN. The above example became

```
y asa x > z
  where
    x = 1 fby x + 2*y + 3;
    y = 0 fby y + 1;
  end
```

(The separate definitions of **first x** and **next x** are combined into a single definition of **x** that uses **fby** (followed by), an operator that works rather like Lisp's **cons**. Also, we adopted Landin's **wherrec**, calling it **where**. The operator **asa** means **as\_soon\_as**, of course.) The implementation had to be changed (which was done by Tony Faustini, who was then a PhD student of Bill Wadge) by adding an initial pass that converted the new Lucid programs into the old ones. Ostrum's implementation was still buried in there. At this point we decided (perhaps misguidedly) to keep the name "Lucid" and not call it "Lucid 2." We thought that the syntax of the language would not change substantially any more. And anyway, we liked the name.

Since this was to be the final form of Lucid, Bill Wadge and I wrote a book about it [26].

### *Space*

By now it was clear that "contexts" held the key to the implementation of Lucid. Nevertheless, when we wanted to add arrays to Lucid we tried for a long time to avoid making contexts more complicated, and instead just stay with time and place. We investigated various algebras of arrays, some of which were very attractive to us, but they just complicated the language too much and threatened to be difficult to implement efficiently. Eventually, we capitulated to the following elegant solution: Contexts were just extended to indicate the position in the array of the value in question, i.e., the "subscripts." Semantically, the values of variables became sequences (in "time") of sequences in space. Operationally, we only ever needed to calculate and store the values for particular contexts, just as we did before when contexts were simpler.

We felt that this solution was too simple and probably would turn out to be hopelessly inadequate, but, in fact, it has turned out to be very powerful yet elegant.

Since contexts were extended, we introduced new versions of the context-dependent operators, **first**, **next**, and **fby**. These are **initial**, **succ** (successor), and **sby** (succeeded by). Using them we can write expressions like

```
bubbled
  where
    bubbled = ORIG fby cond
      iseod bubbled : eod;
      iseod successor : max;
      maxBigger : successor;
      default : max;
```



```

        end;
    max = bubbled sby cond
        iseod successor : eod;
        maxBigger : max;
        default : successor;
    end;
    successor = succ bubbled;
    maxBigger = max > successor;
end

```

which gives us the sequence of arrays, starting with the first array in the sequence of arrays **ORIG** (there is probably only one array in the sequence), formed by repeatedly performing bubblesort-like scans. The array **ORIG** can be of any size; the value **eod** indicates the end of the data. The scans can be done in parallel - they can overlap because each one can proceed as long as it doesn't pass the previous one; education ensures that that doesn't happen.

We begin to see that Lucid programs, especially ones using space, can embody a lot of implicit parallelism.

### *Intensionality*

It was clear to us that the language was very concise and elegant and that the computing model, education, was very powerful. Nevertheless, to observers it probably all seemed rather ad hoc. There was no unifying concept that made sense of it all, apart from the mathematical semantics. It was difficult to explain the language operationally. We used statements like "the values of variables are infinite sequences, but don't think of them as infinite sequences - think of them as changing." Then Bill Wadge came across *Intensional Logic* and it all became clear.

Intensional logic is a logic in which the meanings of expressions and formulas depend on implicit contexts [24]. That is exactly what happens in Lucid. We cannot understand the meaning of, or evaluate, an expression **E** unless we know what context is involved. Technically speaking, the meaning of **E** is a function from contexts to values. In intensional logic this function is called an *intension*. (The value of an intension for a particular context is called an *extension*.)

With this insight, we saw that Lucid is a way to do *intensional programming* - essentially a new programming paradigm [13]. This insight has not invalidated anything we have done, but it has suggested ways forward that we were not thinking of already. It has given us a terminology and a mathematical basis that is reassuring, comforting,

and encouraging. Also, it has given us a way to consistently explain the language. We don't have to say that addition, for example, applies pointwise to infinite sequences. Addition just adds the values of the operands in the context in question.

Intensionality clears up the confusion.

### *Operator Nets*

Back in the late seventies, we could see that there was some connection between Lucid and dataflow, especially dynamic dataflow, but there were some incompatibilities. Bill Wadge and I even wrote a paper "Some Common Misconceptions About Lucid" [8] in which two of the misconceptions we described were "Lucid is a Dataflow Language," and "Lucid is not a Dataflow Language." We were clearly ambivalent about dataflow being relevant to Lucid.

The positive things about dataflow were its absence of control flow (relying instead on data-dependencies) and its asynchronicity and parallelism. The negative things, the problems, were partly syntactic and partly semantic.

The syntactic problems were not crucial. Dataflow researchers always dealt with some sort of dataflow network or dataflow graph, and the ways these graphs were put together seemed to us to involve particular restrictions that were never spelled out or justified. Of course, these were rules that only dataflow people had to live with, and which didn't really affect us, as long as we weren't asked to translate Lucid into dataflow graphs.

The semantic problem was two-fold. First of all, only an operational semantics was given, rather than a mathematical semantics. Secondly, that operational semantics was data-supply driven. We have seen that Lucid implementations are demand-driven, not supply-driven. Some dataflow researchers, notably Pingali [23], have recognised the inefficiency or wastefulness of supply-driven computation. Rather than switch to the demand side, there have been attempts, which we find less than convincing, to show that demand-driven programs can be translated into equivalent supply-driven programs, thereby showing the superiority of the supply side. (On the other hand, Jaggan Jagannathan, one of my PhD students, who was working with me at SRI, showed in his PhD thesis the superiority of the demand-side [14].) There currently seems to be a truce, with the supply side smugly claiming victory and the demand side not believing it and, instead, running practical applications that it smugly claims can not be matched on the supply side.

In the midst of all this I came up with a graphical language that I called *operator*



*nets* [4]. The language is very simple, like dataflow graphs but with none of the above restrictions. It also has a mathematical semantics. Moreover, it is exactly equivalent to Lucid, with the translations from Lucid programs to operator nets, and vice versa, being trivial.

Operator nets can be thought of as the visual forms of Lucid programs. In fact, it is possible to program completely in this visual way, and a PhD student of Tony Faustini, Bill Mitchell, produced a dissertation based partly on a distributed implementation of operator nets called vLucid [20,21] (i.e., Visual Lucid). It is hoped that programming visually will temper the conciseness of Lucid programs.

### *The Education Engine*

For some time we tried to popularize the term "demand-driven dataflow," because that is what we considered we had: a computation model that worked like dataflow except that demands have to flow "backwards" before the data flows forwards.

We failed. It was pointed out to us (by Ian Watson) that people considered the term "demand-driven dataflow" to be an oxymoron, since they felt that "dataflow" was synonymous with "supply-driven." We had to find a different term. Bill Wadge and I found the very appropriate word "eduction" in the (enormous) Oxford English Dictionary: it means "The action of drawing forth, eliciting, or developing from a state of latent, rudimentary, or potential existence; the action of educating (principles, results of calculation) from the data."

(I particularly like an early use of the word (by Hale in 1678) : "The most ancient Atheistick Hypothesis was the Education of all things .. out of Matter.")

At the time I was working at SRI International designing a multiprocessor architecture for Lucid. (Ian Watson was there giving me advice for a month or so.) The machine became known as the Education Engine [1]. Lucid directly as its assembly language. It has a pool of processors, an arbitrary one of which is grabbed whenever a basic operator needs to be applied to the values, in some context, of its operands.

while we were working with an emulator, the machine was discovered to be naturally fault tolerant. Because values are calculated from their definitions whenever they are needed but are not stored, the machine tolerates loss of stored values. This natural fault tolerance can also be enhanced, to cover a wider variety of faults [16].

The machine has never been implemented in hardware, but it has been implemented in software, and that software implementation forms the basis of GLU, which will be

described in the next section.

The Education Engine, like dataflow machines, is fine-grained, which would limit its performance if it were ever built. The software implementation of the Education Engine used by GLU, on the other hand, is coarse-grained. (GLU is short for Granular Lucid.)

Jagannathan developed an architecture and model of computation called eazyflow, a hybrid of eager and lazy dataflow [15]. The eazyflow architecture is a modification of the Education Engine architecture [1].

We will now start to consider more recent developments in Lucid.

### *Parallelism*

We have seen that there is much implicit parallelism in Lucid programs. In a sense, parallelism does not have to be expressed explicitly because it is the norm: expressions can be computed in parallel as long as doing so does not conflict with data-dependencies. Be that as it may, it is still worth the effort of writing parallel programs, by which I mean writing programs that have a great deal of implicit parallelism because the data-dependencies are limited [2]. We can take our cues from standard parallel techniques. For example, we have found it very fruitful to consider tournament computations.

A tournament computation to sum up  $2^k$  numbers in an array  $A$  would do  $2^{k-1}$  additions, in parallel, of adjacent numbers, giving  $2^{k-1}$  values. These are then added in pairs in the same way, giving  $2^{k-2}$  values. In  $k$  steps we have one number: the sum. The computation tree starts with the numbers in the original array  $A$  (the leaves) and proceeds to the single number (the root).

It is relatively simple to program this method in Lucid; I did it in 1988. It can be done as follows:

```
level asa now eq k
```

```
  where
```

```
    level = A fby left_ancestor(level) + right_ancestor(level);
```

```
  end
```

It requires a space dimension: we compute a sequence in "time" of shorter and shorter arrays in "space:" the levels in the computation tree. The predefined time-varying constant **now** indicates what is the time context: it takes on the values 0, 1, 2, .. at time



points 0, 1, 2, ... The  $i$ -th elements, in space, of the array-valued expressions `left_ancestor(level)` and `right_ancestor(level)` denote the two elements of the  $i$ -th pair of elements of `level`. (The pairing is not done explicitly in a data structure. We know which are the elements of the pair by their space positions, as indicated below.)

The functions `left_ancestor` and `right_ancestor` can be defined as follows

```
left_ancestor(a) = a atspace (2*here);
right_ancestor(a) = a atspace (2*here + 1);
```

As with `now`, the predefined space-varying constant `here` indicates what is the space context. The operator `atspace` picks out elements from its left argument that are determined by the values of the right argument. The  $i$ -th element (in space) of `a atspace (2*here)` (i.e., the value in space context  $i$ ) is the value of `a` in space context  $2*i$ , as required.

The operator `atspace` is very useful, and is simple to implement eductively. It would be disastrous to try to implement such an operation in dataflow; the operator is the epitome of demand-driven evaluation. A purely supply-driven evaluation would waste an enormous amount of computation.

The tournament program induces very few data-dependencies: there is just very little there, syntactically. The program implicitly expresses exactly the parallel computation we have talked about.

When it comes to exploiting that parallelism, however, there is a problem. If we use the Education Engine, the granules of parallelism (individual additions) are so small that communication overhead swamps the computation. What we need are larger granules. Jagannathan realized that the larger granules could come from larger operations. Instead of `A` having, say,  $2^{20}$  elements at  $2^{20}$  space points, we can have the elements in  $2^{10}$ -sized chunks at  $2^{10}$  space points. The chunks can all be summed, in parallel, by  $2^{10}$  calls of a C function, and then the resulting  $2^{10}$  values can be summed by a tournament [19]. The tournament might not be very fast, but the simultaneous summings of the chunks would be impressive. (Instead of the tournament, the C function could be used one last time.)

This idea has led to GLU, a version of Lucid in which functions written in C can be utilized [18]. These functions are considered to be basic operators like `+` as far as the Education Engine is concerned (the Education Engine implemented in software, that is) and processors will be grabbed from the processor pool to execute them. The "processors" will be workstations on the internet, for example, and the size of the granules -

the sequential computations of the C function - will render the communication latency insignificant.

GLU has been introduced as a way of introducing C into Lucid programs, in order to increase the granularity of the computation. We can also get to GLU from the other direction: start with a C program and try to introduce parallelism. Essentially what happens is that a layer or veneer of Lucid is added to the C program (after it has been sanitized and expressed as side-effect free and global-variable free functions). The Lucid layer introduces (implicit) parallelism and the software implementation of the Education Engine exploits this parallelism by using different processors or workstations, via RPCs - remote procedure calls. The net result is a system that is usable by procedural programmers, especially if the Lucid layer is expressed graphically (because then Lucid may be more palatable to the programmer).

That is the idea, and it works. Evaluation of the GLU system is in progress, and results will be reported in these proceedings [17].

### *User-Defined Dimensions*

In current implementations of Lucid, objects have just having three dimensions. (There are operators `succ_0`, `succ_1`, `succ_2`, `initial_0`, etc.) This is constraining. Admittedly, it is an inadequacy of the implementation, since the mathematical semantics allows an arbitrary number of dimensions. But it is also a limitation of the language. The mathematical semantics says that the dimensions are numbered 0, 1, 2, 3, 4, etc., but it would be useful to just be able to grab a dimension when you needed one without worrying about whether it is already in use. It would be good to be able to invent your own dimension, a sort of scratch dimension that you will use and then later throw away.

It also would be useful to have dimension parameters. It would be nice to be able to write a function that works on a dimension without knowing which dimension it is. A dummy dimension name would be used in the definition of the function and the call of the function would say which dimension to use in place of the dummy name.

Both these features are found in Indexical Lucid, an extension of Lucid proposed by Tony Faustini [12].

### *Higher-Order Functions*

Lucid has never had higher-order functions. We have never actually been against them (though it has been felt that the intensionality of Lucid makes up to a large ex-



tent for their lack) but, rather, have been unable to implement them eductively. (One of the beauties of intensionality is the ease with which one can have eductive implementations of first-order functions, with no environments.) *Eductive* implementations of higher-order functions have always eluded our grasp.

Recently, there has been progress. George Nelan, a PhD student of Tony Faustini, wrote his dissertation on Firstification, a technique for translating higher-order functional programs to first-order programs [22]. (The method will work for Lucid, provided we add types to the language.) At the same time, Bill Wadge has found an eductive implementation of higher-order functions that uses extra space dimensions for new contexts [25]. This is an example of an advance in the language brought upon by the liberating influence of the concept of intensionality.

### *Whither?*

There are many projects that we can tackle. We need to implement the additions just described - higher-order functions and Indexical Lucid. We also need to investigate more the scalability and applicability of GLU technology. It also would be interesting to go back to look in more detail at the program-proving aspects of Lucid that were left by the wayside fifteen years ago.

The history of Lucid has been marked by dramatic advances that were not planned; they just fell out while we were playing with the semantics. I am sure that we are in for more of this serendipity.

### *References*

1. Ashcroft, E.A., "Eazyflow Architecture," in Selected Reprints on Dataflow and Reduction Architectures, S. S. Thakkar, Editor, IEEE Computer Society, 1987.
2. Ashcroft, E. A., "Parallel Programming in Lucid," 3rd International Symposium on Lucid and Intensional Programming, Kingston, Ontario, 1990.
3. Ashcroft, E. A., Faustini, A. A., and Jagannathan, J., "An Intensional Language for Parallel Applications Programming," in Parallel Functional Languages and Compilers, B. Szymanski, Editor, Addison Wesley, 1991.
4. Ashcroft, E.A. and Jagannathan, R., "Operator Nets", in Fifth Generation Computer Architectures, North Holland, 1986.
5. Ashcroft, E.A., and Wadge, W.W., "Program Proving Without Tears," Proceedings of the International Symposium on Proving and Improving Programs, pp 99-111, Arc et Senans, July 1975.
6. Ashcroft, E. A. and Wadge, W.W., "Lucid - A Formal System for Writing and Proving Programs", SIAM J. Comput., Vol 5, No. 3, 1976.

7. Ashcroft, E. A. and Wadge, W.W., "Clauses: Scope Structures and Defined Functions in Lucid," Proceedings of the Conference on Principles of Programming Languages, January 1977.
8. Ashcroft, E. A. and Wadge, W.W., "Some Common Misconceptions about Lucid," in SIGPLAN Notices, ACM, pp519-526, July 1977.
9. Ashcroft, E. A. and Wadge, W.W., "Intermittent Assertion Proofs in Lucid," IFIP Congress 1977, pp723-726.
10. Ashcroft, E. A. and Wadge, W.W., "Lucid, A Nonprocedural Language with Iteration", CACM, Vol 20, No. 7, 1977.
11. Faustini, A. A., Ashcroft, E. A. and Jagannathan, R., "A Scalable Parallel Architecture for Exploiting Demand-Driven and Data-Driven Computation," in Supercomputing Systems - Chapter 4, Kartashev and Karteshev, Editors, Van Nostrand Reinhold, 1989.
12. Faustini, A.A., and Jagannathan, R., "Indexical Lucid," 4th International Symposium on Lucid and Intensional Programming, Menlo Park, April 1991.
13. Faustini, A.A. and Wadge, W.W., "Intensional Programming," in The Role of Languages in Problem Solving 2, Boudreaux, J.C., Hamill, B.W., and Jernigan, R., Editors, Elsevier Science Publishers B.V. (North-Holland), 1987.
14. Jagannathan, R., "A Descriptive and Prescriptive Model for Dataflow Semantics," PhD Thesis, University of Waterloo, Ontario, 1988.
15. Jagannathan, R. and Ashcroft, E.A., "Eazyflow: A Hybrid Model for Parallel Processing," Proceedings of the International Conference on Parallel Processing, IEEE-ACM, 1984.
16. Jagannathan, R. and Ashcroft, E. A., "Fault Tolerance in Parallel Implementations of Functional Languages," Twenty First International Symposium on Fault-Tolerant Computing, Montreal, 1991.
17. R. Jagannathan and Dodd, C., "Preliminary Performance of a Parallel GLU Implementation on a Network of Workstations," 5th International Symposium on Lucid and Intensional Programming, in Proceedings of the International Conference on Computer Languages, IEEE, Oakland, April 1992.
18. Jagannathan, R. and Faustini, A.A., "GLU: A System for Scalable and Resilient Large-Grain Parallel Processing," 24th Hawaii International Conference on System Sciences, January 1991.
19. Jagannathan, R and Faustini, A.A., "Tournament Computations in GLU," 4th International Symposium on Lucid and Intensional Programming, Menlo Park, April 1991.
20. Mitchell, W.H., "Distributed Visual Dataflow," PhD Thesis, Arizona State University, 1991.
21. Mitchell, W.H. and Ashcroft, E.A., "Rx for Syntax: Operator Nets for Formal Visual Programming," 4th International Symposium on Lucid and Intensional Programming, Menlo Park, April 1991.
22. Nelan, G.C.Jr., "Firstification," PhD Thesis, Arizona State University, 1991.
23. Pingali, K.K. and Arvind, "Efficient Demand-Driven Evaluation. Part 1," ACM



Transactions on Programming Languages and Systems, April 1985, pp311-333.

24. Thomason R., Editor, "Formal Philosophy, Selected Papers of R. Montague," Yale University Press, 1974.

25. Wadge, W.W., "Higher Order Lucid," 4th International Symposium on Lucid and Intensional Programming, Menlo Park, April 1991.

26. Wadge, W. W. and Ashcroft, E. A., "Lucid, the Dataflow Programming Language," Academic Press U.K., 1985.