

MULTIDIMENSIONAL PROGRAM VERIFICATION: REASONING ABOUT PROGRAMS THAT DEAL WITH MULTIDIMENSIONAL OBJECTS

EDWARD ASHCROFT

*Computer Science and Engineering Department, Arizona State University
Tempe, Arizona 85283, USA
E-mail: ed.ashcroft@asu.edu*

ABSTRACT

The formal system aspects of *Lucid* are shown to have been preserved, even though the language has been extended and now deals with multidimensional objects.

1. Introduction

The language *Lucid*⁷ was first introduced over twenty years ago. It was later seen to be an intensional programming language — probably the first.⁵ The original reason for its introduction was that it was a vehicle for a method of program-proving that seemed to be sadly lacking at the time: programs were considered to be sets of assertions about program variables, and properties of programs could be derived by considering the program assertions to be the axioms from which the properties could be derived by essentially conventional mathematical reasoning, using a few proper axioms about the few special operations (*Lucid* operations) and using a special induction rule that we called *Lucid Induction*. *Lucid* was essentially a formal system.

The language *Lucid* has recently been augmented² by allowing variables to vary in many dimensions, and corresponding new *Lucid* operations have been added. Consequently, more proper axioms are needed, and the *Lucid Induction* rule has to be generalized. This paper will detail these changes, and, in the process, will explain again the *Lucid* formal system, something which has needed to be done since the original, one-dimensional formal system was first introduced, by the author and Bill Wadge, in 1976.³ Details of the language, i.e., *Lucid*, will not be explained here, but the formal system for proving properties of *Lucid* programs will be.

2. The Formal System

We will initially assume that *Lucid* programs are *flat*, i.e., that they consist of sets of definitions, with the definitions being equations. Further, the equations will be assumed not to contain subsidiary definitions.

(These assumptions are almost always violated in practice: Lucid programs are usually *structured*, with definitions within definitions. Nevertheless, it is valuable to start with the flat case, and explain later how structured programs give rise to structured proofs.)

The definitions in a program are considered to be the *axioms* of the formal system that corresponds to the program. From these axioms, theorems are *deduced*, and these theorems thus express derived properties of the variables and functions in the program and of the result of the program. The axioms themselves are, of course, theorems with length-zero deduction sequences or derivations. (It makes sense to consider the definitions in a program to be axioms: they are equations, after all, and are truth-valued, like all properties.) The axioms express basic properties of the variables and functions in the program.

The formal system is the logic that allows theorems to be deduced from other theorems. Lucid uses a *natural deduction* system that is very similar to the natural deduction system for first-order predicate calculus.⁶ The system has two rules of inference (deduction rules) for each logical connective, for introducing and eliminating the connective. (This is in contrast to conventional logical systems which tend to have *one* rule of inference (*modus ponens*) and many theorems or axioms that express properties of the logical connectives.)

In addition to these rules for logical reasoning, the Lucid formal system has theorems or axioms that express properties of the Lucid operations. These properties neatly express the meanings of the operations. For example, in Original Lucid we have the property

$$\text{next}(A \text{ fby } B) = B \quad (1)$$

which, in the new Lucid, becomes a set of properties, one for each dimension. For dimension a , Eq. (1) becomes

$$\text{next}.a(A \text{ fby}.a B) = B \quad (2)$$

All Lucid operations in the new version of the language have to be "focussed" onto a dimension, as was shown in Eq. (2) for *next* and *fby*. In Eq. (2), *next* and *fby* have to be focussed onto the *same* dimension, otherwise the property will not be true. We will see later that some properties require operations to be focussed onto dimensions that are *different*.

There are many properties of Original Lucid like Eq. (1) that become properties of New Lucid when all occurrences of Lucid operations in the property are focussed onto the same dimension. Since these Original Lucid properties can be found in the original Lucid Book,⁷ the corresponding New Lucid properties will not be listed here (but will be invoked when necessary). Instead,

we will concentrate on properties that are not the analogs of Original Lucid properties.

2.1. Properties of first, next, etc.

First, we will list properties that involve two or more different dimensions. (If the dimensions were the same, the statements in question might be New Lucid properties that corresponds to existing Original Lucid properties and are therefore true, or might be false because they correspond to false statements in Original Lucid. (For example, Eq. (4) is true and Eq. (5) is false if $a \neq b$.) In either case, nothing will be gained by discussing the statements here.) Here are some useful new properties:

$$\text{first.a first.b } X = \text{first.b first.a } X \quad (3)$$

$$\text{next.a next.b } X = \text{next.b next.a } X \quad (4)$$

$$\text{first.a next.b } X = \text{next.b first.a } X \quad (5)$$

$$\text{first.a}(X \text{ asa.b } Y) = \text{first.a } X \text{ asa.b first.a } Y \quad (6)$$

$$\text{next.a}(X \text{ asa.b } Y) = \text{next.a } X \text{ asa.b next.a } Y \quad (7)$$

$$\text{first.a}(X \text{ fby.b } Y) = \text{first.a } X \text{ fby.b first.a } Y \quad (8)$$

$$\text{next.a}(X \text{ fby.b } Y) = \text{next.a } X \text{ fby.b next.a } Y \quad (9)$$

$$(X \text{ asa.a } Y) \text{ asa.b } Z = (X \text{ asa.b } Z) \text{ asa.a } (Y \text{ asa.b } Z) \quad (10)$$

$$(X \text{ fby.a } Y) \text{ fby.b } Z = (X \text{ fby.b } Z) \text{ fby.a } (Y \text{ fby.b next.a } Z) \quad (11)$$

$$(X \text{ fby.a } Y) \text{ asa.b } Z = (X \text{ asa.b } Z) \text{ fby.a } (Y \text{ asa.b } Z) \quad (12)$$

2.2. Properties of # and @

In order to fully explain the properties of # and @ we will need to refer to the concept of the *rank* of expressions. Intuitively, the rank of an expression is an indication that the value denoted by the expression varies in at least a certain set of dimensions. That set is the rank. The rank is specified in terms of the ranks of subexpressions. The rank is an upper bound (using set inclusion as the partial ordering) on the set of dimensions in which the value *actually* varies. It is not the *least* upper bound, but the aim is to make it the smallest set that is an upper bound and is syntactically determinable (i.e., that can be calculated at compile-time). Because rank is determined by the *form* of an expression, rank is not referentially transparent. We can have two expressions that are equal (denote the same value) but are of different ranks.

Rank has been considered and defined before for the new Lucid. One definition can be found in the paper by the author in the last ISLIP, ISLIP

94¹ and there is a paper about rank for the new Lucid, by Chris Dodd⁴ in the current volume, i.e., in ISLIP 95. In both cases, the definition needs to be tightened up when it comes to structured Lucid, particularly for programs that contain recursive functions that introduce new dimensions and then make a recursive call *using the new dimension*, but these are rare cases, that no doubt will be handled and cleared up in the near future.

Here are some properties of # and @:

$$(X@.aI)@.bJ = (X@.bJ)@.aI \quad (13)$$

(provided I does not vary in dimension b , J does not vary in dimension a , and a and b are different; that is, $b \notin \text{rank}(I) \& a \notin \text{rank}(J) \& a \neq b$).

$$\#.a @.aI = I \quad (14)$$

$$X@.a \#.a = X \quad (15)$$

Notice that $\text{rank}(X@.a \#.a) = \text{rank}(X) \cup \{a\}$, so, if $a \notin \text{rank}(X)$, Eq. (15) is an equation where the two sides have different ranks.

Defining `realign` as follows

$$\text{realign}.a,b(X) = X@.a \#.b \quad (16)$$

we can easily verify the following properties:

$$\text{realign}.a,b(X)@.by = X@.ay \quad (17)$$

(provided $b \notin \text{rank}(X) \cup \text{rank}(y)$)

$$\text{realign}.a,b(\text{realign}.b,a(X)) = X \quad (18)$$

(provided $a \notin \text{rank}(X)$).

These properties are very useful for proving properties of programs that do not use recursion. They were used, for example, in the new Lucid book to show the correctness of the transpose program that was defined there in terms of `realign`. Also in the new book was a function `runningSum` that used a recursively defined variable. Here is its definition:

```
runningSum.d(y) = s where
    s = y fby.d next.d y + s;
end;
```

In the book it was stated that

$$\begin{aligned} \text{runningSum}.a(\text{runningSum}.b(A)) = \\ \text{runningSum}.b(\text{runningSum}.a(A)) \end{aligned} \quad (19)$$

It was also stated that Eq. (19) could be *proved*. We will do that here, but to do so we will need to use a generalized form of *Lucid Induction*.

2.3. *Lucid Induction Generalized*

The *Lucid Induction* that had been around since *Lucid* was first invented assumed that there was just one dimension to think about and reason about, and that dimension was implicit. (If ever referred to, we called it “time.”) Now that there can be an arbitrary number of dimensions, and the dimensions have names, we need to have an induction rule that works in more than one dimension. In fact, proving Eq. (19) requires such a rule (or, at least, the author has been unable to prove it using the old 1-D *Lucid Induction*).

Here is the generalized form of the *Lucid Induction Rule* for the set of dimensions $\{a, b\}$:

$$\text{first.a } P, \text{ first.b } P, P \rightarrow \text{next.a next.b } P \quad (20)$$

$$P \quad (21)$$

The first two premises are essentially the *base cases* and the last premise is the *induction step*. In order to use the Rule, we must prove a theorem that contains an implication, i.e., contains the connective \rightarrow . Well, can't we use the \rightarrow introduction rule from the natural deduction system? Unfortunately, *No*. The \rightarrow Introduction Rule is the only rule in the conventional natural deduction system, apart from the rules for negation, that does *not* carry over to the *Lucid* natural deduction system. It is easy to see why: \rightarrow is pointwise implication — $A \rightarrow B$ means that in every context where A is true B is true also. That is saying much more than “the truth of A in all contexts implies the truth of B in all contexts,” but the \rightarrow Implication Rule (the Deduction Theorem) says that the latter implies the former. It therefore obviously is not true.

An astute reader will have noticed that the original *Lucid Induction Rule*, of which the multidimensional rule is a generalization, must contain \rightarrow also. How, then, had it been possible to use that rule if \rightarrow could not be introduced? This gets us to one of the most subtle and clever results in the 1976 paper. (I must admit that the cleverness was Bill Wadge's and not the author's.) The deduction theorem can be recovered, but at the expense of weakening the = Elimination Rule (the rule that allows the replacement of an occurrence of the left-hand-side of an equality by the right-hand-side, or *vice versa*), in the following way. To prove $A \rightarrow B$, we must first assume A and then manage to prove B *without ever replacing an expression by an equal expression WITHIN THE SCOPE OF A LUCID OPERATION THAT IS FOCUSSED ON SOME DIMENSIONS THAT WE ARE TRYING TO DO INDUCTION OVER*.

This is obviously an = Elimination Rule that is geared to the generalized form of *Lucid Induction* for the new *Lucid*. The weakened = Elimination Rule in the 1976 paper disallowed substitution within the scope of *any* *Lucid* operation. Of course, at that time all *Lucid* operations were focussed on the same, implicit, dimension. Now, with generalized *Lucid Induction*, we can do induction over specific dimensions, and we only disallow substitutions within the scope of *Lucid* operations that are focussed on those specific dimensions.

If we can prove B in this way, and we don't use any *Lucid* rules that are focussed on the dimensions we are doing induction over, we can "discharge the assumption", A , and conclude $A \rightarrow B$ with no assumption A .

3. A Generalized *Lucid Induction* Proof

We now prove Eq. (19). To do this we need three properties of *runningSum* that are relatively straightforward to prove, namely

$$\text{next.i runningSum.j}(X) = \text{runningSum.j}(\text{next.i } X) \quad (22)$$

$$\text{runningSum.i}(X+Y) = \text{runningSum.i}(X) + \text{runningSum.i}(Y) \quad (23)$$

$$\text{next.i runningSum.i}(X) = \text{runningSum.i}(X) + \text{next.i } X \quad (24)$$

(The proofs will not be given here.)

We will use 2-D *Lucid Induction* to prove Eq. (19). That is, Eq. (19) is the property P in the induction rule. The three premises of the 2-D Rule are then

$$\begin{aligned} \text{first.a}(\text{runningSum.a}(\text{runningSum.b}(A))) = \\ \text{runningSum.b}(\text{runningSum.a}(A)) \end{aligned} \quad (25)$$

$$\begin{aligned} \text{first.b}(\text{runningSum.a}(\text{runningSum.b}(A))) = \\ \text{runningSum.b}(\text{runningSum.a}(A)) \end{aligned} \quad (26)$$

$$\begin{aligned} \text{runningSum.a}(\text{runningSum.b}(A)) = \\ \text{runningSum.b}(\text{runningSum.a}(A)) \rightarrow \\ \text{next.a next.b runningSum.a}(\text{runningSum.b}(A)) = \\ \text{runningSum.b}(\text{runningSum.a}(A)) \end{aligned} \quad (27)$$

We will prove each of these, and thus conclude

$$\text{runningSum.a}(\text{runningSum.b}(A)) = \text{runningSum.b}(\text{runningSum.a}(A))$$

Since *first* and *next* distribute through pointwise operators like $+$ and $=$, to prove Eq. (25), we will actually prove

$$\begin{aligned} \text{first.a runningSum.a}(\text{runningSum.b}(A)) = \\ \text{first.a runningSum.b}(\text{runningSum.a}(A)) \end{aligned} \quad (28)$$

and we will do this by using 1-D Lucid Induction on dimension b . The two premises of *this* induction are

$$\begin{aligned} & \text{first.b first.a runningSum.a(runningSum.b(A))} = \\ & \text{first.b first.a runningSum.b(runningSum.a(A))} \end{aligned} \quad (29)$$

$$\begin{aligned} & \text{first.a runningSum.a(runningSum.b(A))} = \\ & \text{first.a runningSum.b(runningSum.a(A))} \rightarrow \\ & \text{next.b first.a runningSum.a(runningSum.b(A))} = \\ & \text{next.b first.a runningSum.b(runningSum.a(A))} \end{aligned} \quad (30)$$

(Actually, several steps were omitted here, of the sort that got us from Eq. (25) to Eq. (28).

To prove Eq. (29) we proceed as follows:

$$\begin{aligned} & \text{first.b first.a runningSum.a(runningSum.b(A))} \\ & = \text{first.b first.a runningSum.b(A)} \end{aligned} \quad (31)$$

$$= \text{first.a first.b runningSum.b(A)} \quad (32)$$

$$= \text{first.a first.b } A \quad (33)$$

$$= \text{first.b first.a } A \quad (34)$$

$$= \text{first.b first.a runningSum.a(A)} \quad (35)$$

$$= \text{first.a first.b runningSum.a(A)} \quad (36)$$

$$= \text{first.a first.b runningSum.b(runningSum.a(A))} \quad (37)$$

$$= \text{first.b first.a runningSum.b(runningSum.a(A))} \quad (38)$$

□

To prove Eq. (31) and Eq. (35), we use the definition of `runningSum.a`, unfolding and folding, respectively. (This is substituting equals for equals within the scope of `first.b`, but since we are not intending to use the \rightarrow Elimination Rule for Eq. (29) it is OK to do so. This same comment applies to all the steps in the proof of Eq. (28).) To prove Eq. (33) and Eq. (37), we use the definition of `runningSum.b`, unfolding and folding, respectively, and the rest of the steps all use Eq. (3), for appropriate dimensions.

To prove Eq. (30) we have to introduce a \rightarrow connective, so we make an assumption and then are careful not to substitute within the scope of a Lucid operation focussed on dimension b . (We intend to use Lucid Induction on dimension b .)

We assume the left hand side of the pointwise implication, namely

$$\begin{aligned} & \text{first.a runningSum.a(runningSum.b(A))} = \\ & \text{first.a runningSum.b(runningSum.a(A))} \end{aligned} \quad (39)$$

We now proceed as follows:

$$\begin{aligned}
 & \text{next.b first.a runningSum.a}(\text{runningSum.b}(A)) \\
 &= \text{first.a next.b runningSum.a}(\text{runningSum.b}(A)) \quad (40) \\
 &= \text{first.a runningSum.a}(\text{next.b runningSum.b}(A)) \quad (41) \\
 &= \text{first.a runningSum.a}(\text{runningSum.b}(A) + \text{next.b } A) \quad (42) \\
 &= \text{first.a runningSum.a}(\text{runningSum.b}(A)) + \\
 &\quad \text{first.a runningSum.a}(\text{next.b } A) \quad (43) \\
 &= \text{first.a runningSum.b}(\text{runningSum.a}(A)) + \\
 &\quad \text{first.a next.b runningSum.a}(A) \quad (44) \\
 &= \text{first.a}(\text{runningSum.b}(\text{runningSum.a}(A)) + \\
 &\quad \text{next.b runningSum.a}(A)) \quad (45) \\
 &= \text{first.a next.b runningSum.b}(\text{runningSum.a}(A)) \quad (46) \\
 &= \text{next.b first.a runningSum.b}(\text{runningSum.a}(A)) \quad (47)
 \end{aligned}$$

We have obtained the right hand side of the pointwise implication. Since we have made an assumption (the left hand side), and we intend to use generalized induction focussed on dimension b , we must be sure that we haven't substituted equals for equals within the scope of a *Lucid* operation focussed on dimension b . To prove Eq. (40), we used Eq. (5) and the substitution was not within the scope of any *Lucid* operation. To prove Eq. (41), we used Eq. (22) and the substitution was only within the scope of *first.a*, so that is OK. To prove Eq. (42), we used Eq. (24), but the substitution was within the scope of the *function* *runningSum.a* as well as the operation *first.a*! This appears to violate the weak = Elimination rule, which only talks about *operations*. It is easy to see, however, that substituting within the scope of the function is OK: unfold the function call and do the substitution within the arguments (no operation focussed on b is introduced by the unfolding) and then fold it up again; all this is done within the scope of *first.a*.

(To deal with functions, we probably should introduce the concept of the dimensions that affect a function. These can be found from the dimensions in the rank of the the expression that is the right hand side of the function's definition. In this case, the rank of

```

s  where
    s = y fby.d next.d y + s;
end;

```

is $\{d\} \cup \text{rank}(y)$, and the dimensions that affect the function are just $\{d\}$, i.e., the dimension that the function is focussed on. That means that as far as an induction proof is concerned, the function *runningSum* is like a *Lucid* opera-

tion. Other functions might be different: they might be affected by dimensions that are global to the function.)

To prove Eq. (43), we used Eq. (23) and the substitution was only within the scope of `first.a`. To prove Eq. (44), we used our assumption and also Eq. (22). In both cases, the substitutions were only within the scope of `first.a`. To prove Eq. (45), we used an Original Lucid property, the fact that `first` and other Lucid operations distribute through pointwise operations. The substitution was not within the scope of any Lucid operation. To prove Eq. (46), we used Eq. (24) and the substitution was only within the scope of `first.a`. To prove Eq. (47), we used Eq. (5) and the substitution was not within the scope of any Lucid operation.

Since all the substitutions were not within the scopes of Lucid operations, we can discharge our assumption, giving us the second premise and therefore the desired result, namely Eq. (28).

By symmetry, we also get Eq. (26).

To complete the proof of Eq. (19) we must prove the induction step, Eq. (27). To do this we eventually will have to assume Eq. (19), the left hand side of the pointwise implication, but we will delay doing that as long as we can, so that we are not restricted in the use of substitutions. (We could have delayed making an assumption in the proof of Eq. (30), and avoided making all the checks on admissibility of substitutions, but it was interesting to see that we didn't *need* to delay making the assumption. In *this* case, we *have to* delay making the assumption.)

We proceed, then, to prove the right hand side of the implication, namely

$$\begin{aligned} \text{next.a next.b runningSum.a}(\text{runningSum.b}(A)) = \\ \text{next.a next.b runningSum.b}(\text{runningSum.a}(A)) \end{aligned} \quad (48)$$

as follows, starting with the left hand side of Eq. (48).

$$\text{next.a next.b runningSum.a}(\text{runningSum.b}(A)) \quad (49)$$

$$= \text{next.b next.a runningSum.a}(\text{runningSum.b}(A)) \quad (50)$$

$$\begin{aligned} = \text{next.b}(\text{runningSum.a}(\text{runningSum.b}(A)) + \\ \text{next.a runningSum.b}(A)) \end{aligned} \quad (51)$$

$$\begin{aligned} = \text{next.b runningSum.a}(\text{runningSum.b}(A)) + \\ \text{next.b next.a runningSum.b}(A) \end{aligned} \quad (52)$$

$$\begin{aligned} = \text{runningSum.a}(\text{next.b runningSum.b}(A)) + \\ \text{next.a next.b runningSum.b}(A) \end{aligned} \quad (53)$$

$$= \text{runningSum.a}(\text{runningSum.b}(A) + \text{next.b } A) +$$

$$\text{next.a}(\text{runningSum.b}(A) + \text{next.b } A) \quad (54)$$

$$= \text{runningSum.a}(\text{runningSum.b}(A)) + \text{next.b runningSum.a}(A) + \text{next.a runningSum.b}(A) + \text{next.a next.b } A \quad (55)$$

We now make a completely symmetrical argument, starting with the right hand side of Eq. (48):

$$\text{next.a next.b runningSum.b}(\text{runningSum.a}(A)) \quad (56)$$

$$= \text{next.a}(\text{runningSum.b}(\text{runningSum.a}(A)) + \text{next.b runningSum.a}(A)) \quad (57)$$

$$= \text{next.a runningSum.b}(\text{runningSum.a}(A)) + \text{next.a next.b runningSum.a}(A) \quad (58)$$

$$= \text{runningSum.b}(\text{next.a runningSum.a}(A)) + \text{next.b next.a runningSum.a}(A) \quad (59)$$

$$= \text{runningSum.b}(\text{runningSum.a}(A) + \text{next.a } A) + \text{next.b}(\text{runningSum.a}(A) + \text{next.a } A) \quad (60)$$

$$= \text{runningSum.b}(\text{runningSum.a}(A)) + \text{next.a runningSum.b}(A) + \text{next.b runningSum.a}(A) + \text{next.b next.a } A \quad (61)$$

The justifications for the two arguments are as follows. To prove Eq. (50), we use Eq. (4). To prove Eq. (51) and Eq. (57), we use Eq. (24). To prove Eq. (52) and Eq. (58), we use the fact that *next* distributes through *+*. To prove Eq. (53) and Eq. (59), we use Eq. (22) and Eq. (4). To prove Eq. (54) and Eq. (60), we use Eq. (24), twice. To prove Eq. (55) and Eq. (61), we use Eq. (23), Eq. (22), and the distribution of *next* through *+*.

We now assume Eq. (19), and using it and the fact that

$$(A + B) + C = (A + C) + B$$

and Eq. (4) we see that Eq. (55) becomes Eq. (61), and *no substitutions were made within the scope of any Lucid operation*. That means that Eq. (49) and Eq. (56) are equal. That is, we have proved Eq. (48), under the assumption of Eq. (19). We now discharge the assumption, and we have proved Eq. (27).

Thus, by 2-D *Lucid* Induction, we have proved Eq. (19), as promised.

3.1. Bonus

It is interesting that

$$(A - B) - C = (A - C) - B$$

is true also, which implies that a similar proof establishes that

$$\begin{aligned} \text{runningDiff}.a(\text{runningDiff}.b(A)) = \\ \text{runningDiff}.b(\text{runningDiff}.a(A)) \quad - \end{aligned} \quad (62)$$

where `runningDiff` is defined like `runningSum` but with `-` in place of `+`.

4. Proofs in Structured Lucid

We have been assuming that the Lucid programs we are considering are flat, but, in fact, the function `runningSum` does not have a flat definition. When programs are structured, the only affect on proofs is that the assertions that are proved have to be considered to be true within specific regions of the program. In the example proof, all the properties we established were true at the outermost level.

In general, properties true at one level are true at deeper levels (barring name clashes). Properties true at one level are also true at higher levels, providing there are no local variables, functions, or dimensions mentioned in the property that are meaningless at the higher levels.

5. Conclusion

We have shown that proofs can still be carried out for Lucid programs, even though they are now dealing with multidimensional objects. The properties of these multidimensional objects are often stated in terms that require knowledge of the ranks of expressions, even though programs themselves can not refer to rank.

A generalized Lucid Induction principle is described, and a detailed proof using it is shown, in order to illustrate and explain the dimension-specific reasoning that has to be employed.

6. References

1. E.A. Ashcroft, Multidimensional Declarative Programming, In *Proceedings Seventh International Symposium on Lucid and Intensional Programming*, SRI International, USA, September 1994. (Available from <http://www.csl.sri.com/lucid/ISLIP94/electronic-proceedings.html>)
2. E.A. Ashcroft, A.A. Faustini, R. Jaggannathan, W.W. Wadge, *Multidimensional Programming*, Oxford University Press, 1995.
3. E.A. Ashcroft and W.W. Wadge, Lucid — A Formal System for Writing and Proving Programs, *SIAM J. Comput.*, Vol 5, No. 3, September 1976.
4. C. Dodd, Rank Analysis in the GLU Compiler, *This volume*.

5. A.A. Faustini and W.W.Wadge, Intensional Programming, In *The Role of Languages in Problem Solving 2*, J.C. Boudreaux, B.W. Hamill, and R. Jenigan, editors, Elsevier Science Publishers B.V. North-Holland, 1987.
6. Z. Manna, *Introduction to Mathematical Theory of Computation* McGraw-Hill, New York, 1974.
7. W.W. Wadge and E.A. Ashcroft, *Lucid, the Dataflow Programming Language*, Academic Press, U.K., 1985.