Acta Informatica 9, 217–241 (1978)

Acta Informatica

Design and Correctness of a Compiler for a Non-Procedural Language*

Christoph M. Hoffmann

Department of Computer Science, Purdue University, West Lafayette, IN 47907, USA

Summary. Design and correctness proof of a compiler for Lucid, a non-procedural proof-oriented programming language, are given. Starting with the denotational semantics of Lucid, an equivalent operational semantics is derived, and from it the design of compiling algorithms. The algorithms are proved to compile correctly a subset of the language. A discussion of the design choices and of the subset restrictions gives insight into the nature of Lucid as well as into the problem of compiling related non-procedural languages.

1. Introduction

Proving the correctness of a compiler is an important and non-trivial problem. Without a correct compiler, no program compiled can be fully trusted, even though the program was proved correct. Furthermore, as compilers for all but very small languages are of substantial size, a correctness proof lends credibility to program proving as a viable discipline applicable to practical work.

This paper is concerned with the design and correctness proof of an efficient compiler for a very high-level non-procedural programming language, and combines techniques from various disciplines of Computer Science to shape a reliable piece of software. As the syntax of the source language is almost trivial, syntax directed methods for compiling were of little help. New techniques had to be developed which are reasonably general in nature and should be applicable to a broad class of non-procedural languages.

A first version of the compiler has been operational now for over two years. The subsequent work on a proof for it in turn affected the design of the algorithms. This development illustrates the maxim that programming and program proving are complementary efforts which best are done in parallel.

* Work supported by the National Research Council of Canada

0001-5903/78/0009/0217/$05.00

Proving correctness of a compiler is difficult for several reasons. To begin with, it requires a formal model of the semantics of the source language as well as the object language, which, moreover, has to be convenient for both proof purposes and implementation strategies. Without such models a proof cannot be stated. If the models are difficult to analyze mathematically, a proof will be awkward and difficult. Therefore, it is understandable that much of the previous work in this area was done for compilers of languages based on the Lambda Calculus [14, 17] or for small languages isolating a few of the constructs present in imperative languages [7, 15, 18]. Present trends in the work on data structures and programming language design [10, 11] seem to indicate an awareness of the necessity of a formal model of semantics.

Another difficulty in proving compilers correct, as recognized by Morris [18], is to find a compiler design structured in such a way that a proof can be modularized, thereby reducing the complexity of the task. Not surprisingly, this maxim has analogous formulations which have long since been principles of software design.

The language to be compiled is Lucid, and was developed by Ashcroft and Wadge [3, 5]. Its motivation is to bridge the gap felt to exist between practical demands on language constructs, as vehicles for expressing algorithms, and constructs amenable to rigorous mathematical analysis. Unlike other approaches, e.g. [20], Lucid uses the same denotation for writing and proving properties of programs, thus it is, at the same time, a formal proof system and a programming language. Previously published work has emphasized this point strongly [2, 4].

From a programming point of view it is interesting to note that the flavor of this language is not unlike that of recently proposed data-flow languages, e.g. [8, 13], in which, as the name suggests, the flow of values through a network of processing nodes is specified, and an explicit notion of flow of control is absent. The reason for this similarity may be deduced from the fact that the meaning of a variable in Lucid does not depend on its context in the program. This property is shared by data-flow languages [8] and, at the same time, simplifies the notion of environment on which proofs of program properties depend crucially.

A Lucid program is a set of assertions about sequences of data objects. Understanding these (infinite) sequences as streams, it is the task of the compiler to analyze implied data dependencies and to deduce from this information a strategy for a coordinated evaluation of the various streams. The language and a compilable subset are defined in Section 2, both formally and informally.

Perhaps the first difficulty encountered in compiling Lucid stems from the fact that the semantics of the language is defined denotationally. Since a direct implementation of the fix-point operator is far from being efficient, an equivalent operational understanding had to be sought. This is derived in Section 3.

The compilation algorithms are developed first for very simple programs and analyzed for correctness. The correctness proof of the object code employs Floyd's method [9] which was found to be very well-suited because of the simple control structure of the generated code. Because of the modularity of the source language constructs, easy extensions of the algorithms are possible which

enable a compilation of the full subset. These are described subsequently. A discussion of the language restrictions for compilability concludes the paper, and sheds further light on the design choices.

2. Lucid and a Compilable Subset

We define the source language Lucid and a subset acceptable to the compiler. Reasons for these subset restrictions are discussed in Section 6. Not included in this presentation is the definition of nested blocks. As explained in detail in [12], this somewhat complicated construct can be handled quite easily by a recursive extension of our algorithms.

A program in Lucid can be thought of as a set of commands describing an algorithm in terms of assignments and loops, and, at the same time, as a set of equations which are assertions about the results and effects of the program. A major problem in achieving this is to find a way in which an assignment statement can be viewed as a mathematical equation. While this is always possible for the first assignment to a variable, reassignments such as $C \leftarrow C+1$ cannot be viewed as equations. Clearly any reassignment can be eliminated by introducing new variables in the absence of iteration; in loops, however, it has to be accomplished differently. To this effect, Lucid distinguishes between the initial value of a variable in a loop (first C), the value of a variable during the current iteration (C), and the value during the next iteration (next C). Then the pair of equations

$$\text{first } C = 0,$$
$$\text{next } C = C+1$$

would imply a loop in which C has initially the value 0 and is to be incremented by 1 each time the loop is repeated. The fby operator (pronounced "followed by") is used to abbreviate such two part definitions, e.g.

$$C = 0 \text{ fby } C+1.$$

Abstracting, we can view C as the (infinite) sequence $\langle 0, 1, 2, 3, \ldots \rangle$. Of course, the definition of a variable may involve others. So, e.g.,

$$\text{first } C = 0 \quad \text{next } C = C+1,$$
$$\text{first } Y = 1 \quad \text{next } Y = Y + 2 * C + 3$$

or, equivalently,

$$C = 0 \text{ fby } C+1,$$
$$Y = 1 \text{ fby } Y + 2 * C + 3$$

defines the next value of Y in terms of the current values of Y and C. Here it is verified that Y is the sequence of perfect squares $\langle 1, 4, 9, 16, \ldots \rangle$.

In addition, there is a binary operator asa (pronounced "as soon as") which extracts values from a loop:

$$C = 0 \text{ fby } C + 1,$$
$$Y = 1 \text{ fby } Y + 2 * C + 3,$$
$$R = C \text{ asa } Y > 10.$$

The equation for R 'extracts' the value of C of the loop iteration during which the value of Y is greater than 10 for the first time (i.e. 3), thus R is the (constant) sequence $\langle 3, 3, 3, \ldots \rangle$. In a sense, then, an asa expression is a control expression for a loop.

More formally, a variable X is interpreted as a mapping α from the natural numbers N into a set of values V which includes the special value $\perp$ (pronounced "undefined"). Instead of writing $\alpha(t)$ we write α_t.

There are four *special operators*, first, next, fby, asa. They are interpreted as follows.

Definition. Let α and β be mappings from N into V, then

$$(\text{first } \alpha)_t = \alpha_0,$$
$$(\text{next } \alpha)_t = \alpha_{t+1},$$
$$(\alpha \text{ fby } \beta)_t = \begin{cases} \alpha_0 & \text{if } t = 0 \\ \beta_{t-1} & \text{otherwise,} \end{cases}$$
$$(\alpha \text{ asa } \beta)_t = \begin{cases} \alpha_s & \text{if } \beta_s = \textbf{true},\ \beta_r = \textbf{false} \text{ for all } r < s \\ \perp & \text{otherwise} \end{cases}$$

Besides the four special operators, there are a number of *standard* operators which work 'point-wise', i.e., for the r-ary standard operator ω and mappings $\alpha^{(1)}, \alpha^{(2)}, \ldots, \alpha^{(r)}$ from N into V, then

$$(\omega, \alpha^{(1)}, \ldots, \alpha^{(r)})_t = (\omega, \alpha_t^{(1)}, \ldots, \alpha_t^{(r)}).$$

In the following, we fix the set of standard operators to consist of the operators $*$, /, +, −, **eq**, **ne**, **lt**, **gt**, **le**, **ge**, **not**, **and**, **or**, **if**−**then**−**else**−; and the set V of values to contain all integers and the logical values **true** and **false**. It should be clear that both sets can be enriched by adding other point-wise operators and data values. Such additions do not greatly affect our algorithms and are therefore excluded from consideration.

Expressions are written in the familiar way. Precedence among standard operators is as in ALGOL; the fby and asa operators have lower precedence than any standard operator, with asa taking precedence over fby. Once the interpretation of all variables in an expression E has been fixed, the interpretation of E is given by the above definitions, and is also a mapping from N into V.

A *formula* is an expression formed only from variable names, standard operators, and the operators first and next.

A variable X may be defined in a Lucid program by an equation which is of one of the three forms below

(1) $X = E$,

(2) $X = E$ fby F,

(3) $X = E$ asa F

where E and F are formulae. Restricting the complexity of expressions in this way is no loss of generality, as more complicated expressions can be put into this form using additional variable definitions.

As an analogy, observe that first and next are reminiscent of *car* and *cdr* in LISP, the fby provides a two-part definition in a potentially recursive way, and the asa is analogous to the μ operator in Recursive Function Theory.

We can now give a definition of Lucid programs:

⟨program⟩ ::= **compute** output **where** ⟨block body⟩
⟨block body⟩ ::= ⟨assertion⟩ {; ⟨assertion⟩}* **end**
⟨assertion⟩ ::= ⟨sequence def⟩
| ⟨result def⟩
⟨sequence def⟩ ::= ⟨name⟩ = ⟨expression⟩
⟨result def⟩ ::= **result** = ⟨expression⟩
⟨expression⟩ ::= ⟨formula⟩
| ⟨formula⟩ fby ⟨formula⟩
| ⟨formula⟩ asa ⟨formula⟩

Semantically, a program is an assertion about the sequence output. Other assertions can be made as part of the program defining computational relationships among the variables mentioned. The expression in the result definition defines the name in the compute clause, i.e. output.

Any assignment of interpretations to the variables of a program consistent with the program is a solution of it. The meaning of a program is its minimal (least defined) solution which was shown to exist in [4]. For a more detailed formal definition the reader is referred to that paper.

Example 2.1. As Example of a Lucid program consider

compute output **where**
$C = 0$ fby $C + 1$;
$Y = 1$ fby $Y + 2 * C + 3$;
$N =$ first input;
result $= C$ asa Y **gt** N
end

The program reads the first input as value of N, and iterates a loop which evaluates Y and C until Y is greater than N for the first time. Then the current

value of C is assigned to output, i.e. printed. In effect, since Y is a sequence of square numbers enumerated by C, the program computes the integer square root of its first input.

The compiler performs several source transformations. Among these is the introduction of new variables to simplify expressions to conform with the syntax as given above, and deducing syntactically which sequences are constant. While methods for the former are evident, the latter rests on the following definition of (syntactic) *quiescence.*

Definition. An expression E is *quiescent* if

(a) E is a constant, or

(b) E is a name defined by a quiescent expression, or

(c) $E=(\omega, F_1, \ldots, F_r)$ where ω is a standard operator and the expressions F_i are all quiescent, or

(d) $E=\mathsf{first}\,F$, where F is any expression, or

(e) $E=\mathsf{next}\,F$, where F is a quiescent expression, or

(f) $E=F\ \mathsf{asa}\ G$, where F and G are expressions.

Nothing else is quiescent.

It is not difficult to see that for a quiescent expression E the equation $E=\mathsf{first}\,E$ is valid.

Definition. A Lucid program is *compilable* if the following requirements are met:

(1) Every variable name referenced is defined exactly once except for the name input which may not be defined.

(2) There is exactly one result definition.

(3) Each name is either of type integer or boolean and its definition and usage are consistent with its type.

(4) Construct for the program a labelled directed graph as described below. Then the sum of the edge labels of all edges forming a cycle must be negative for every cycle in the graph.

Restrictions (1) to (3) are straight-forward. The construction of the graph for restriction (4) is presented next. As explained in Section 7 of [12], however, the graph is never constructed by the compiler, rather equivalent criteria are used. Nonetheless, the graph construction is useful for visualizing the essence of that restriction which could be put intuitively by requiring that the current value of a variable should not depend on some future value of itself.

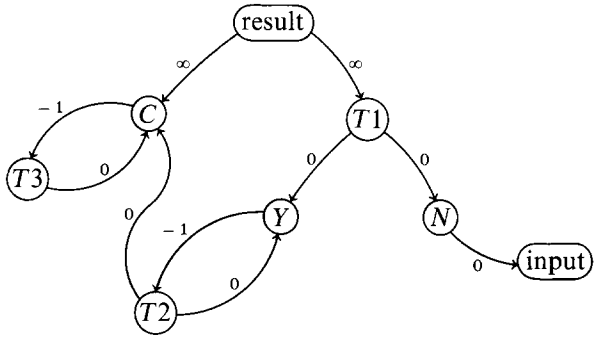
In constructing the graph, assume that additional variables have been defined such that all expressions have been simplified to be in one of the following five forms, where X and Y denote variable names:

(a) $\mathsf{first}\,X$,
(b) $\mathsf{next}\,X$,
(c) $X\ \mathsf{asa}\ Y$,
(d) $X\ \mathsf{fby}\ Y$,
(e) formula not containing any first or next.

The nodes of the graph are now the variables occurring in the new program. If a variable A is defined by

(a) first X, draw an arc from A to X labelled 0;
(b) next X, draw an arc from A to X labelled 1;
(c) X fby Y, draw an arc from A to X labelled 0, and an arc from A to Y labelled -1;
(d) X asa Y, draw an arc from A to X and from A to Y both labelled ∞;
(e) formula without first and next,
draw an arc from A to every variable name occurring in the formula labelled 0.

Example 2.2. The graph for the program of Example 2.1 is as follows after variables T1, T2 and T3 have been introduced in order to simplify all expressions:



Observe that the edge labels for every closed cycle sum up to -1.

The graph constructed in this manner is referred to as (program) *dependency graph.*

3. Derivation of an Operational Semantics

Recall that the meaning of a Lucid program is defined by a fix-point semantics. Since, for computational purposes, it must be related to an operational semantics, expressions are to be manipulated into a different representation, which makes more apparent the connection between the fixpoint interpretation and an equivalent operational understanding of the constructs. To this end two mappings, ϕ_n and τ_n, where $n \geqq 0$, are introduced. At the same time, ϕ_n and τ_n effect a removal of the operators next and first.

Two operators, $:i$ and $\langle i \rangle$, where $i \geqq 0$, are needed which, applied to variables as suffix operations, have the following interpretation.

Definition. Let X be a variable, $\alpha = |X|$ an interpretation of X, i.e. a mapping from N into V. Then, for all t in N,

(1) $(|X:i|)_t = \alpha_i$,

(2) $(|X\langle i \rangle|)_t = \alpha_{t+i}$.

Note the analogy to first and next.

Definition. An expression formed from constants, paretheses, standard operators and variable names to each of which exactly one suffix operator $:i$ or $\langle i \rangle$ is applied is called a *term*. Expressions formed as above but also admitting asa and fby are called *extended terms*.

Example 3.1. The following are terms:

$$5,\ X{:}0*(Y\langle 1\rangle + Y\langle 0\rangle),\quad X\langle 3\rangle.$$

The following is an extended term

$$X\langle 0\rangle \text{ asa } (Y{:}0 \text{ fby } X\langle 2\rangle) \textbf{ eq } 405.$$

The mappings ϕ_n and τ_n, $n \geqq 0$, mapping expressions into extended terms are defined as follows.

Definition.

(a) If X is a variable name, then

$$\phi_n(X) = X{:}n,$$
$$\tau_n(X) = X\langle n\rangle.$$

(b) If ω is an r-ary standard operator, $F^{(1)} \ldots F^{(r)}$ expressions, then

$$\phi_n(\omega, F^{(1)}, \ldots, F^{(r)}) = (\omega, \phi_n(F^{(1)}), \ldots, \phi_n(F^{(r)})),$$
$$\tau_n(\omega, F^{(1)}, \ldots, F^{(r)}) = (\omega, \tau_n(F^{(1)}), \ldots, \tau_n(F^{(r)})).$$

(c) If F is an expression, then

$$\phi_n(\text{first } F) = \phi_0(F) \qquad \tau_n(\text{first } F) = \phi_0(F),$$
$$\phi_n(\text{next } F) = \phi_{n+1}(F) \qquad \tau_n(\text{next } F) = \tau_{n+1}(F).$$

(d) If F and G are expressions, then

$$\phi_0(F \text{ fby } G) = \phi_0(F),$$
$$\tau_0(F \text{ fby } G) = \phi_0(F) \text{ fby } \tau_0(G),$$
$$\phi_{n+1}(F \text{ fby } G) = \phi_n(G),$$
$$\tau_{n+1}(F \text{ fby } G) = \tau_n(G).$$

(e) If F and G are expressions, then

$$\phi_n(F \text{ asa } G) = \tau_0(F) \text{ asa } \tau_0(G),$$
$$\tau_n(F \text{ asa } G) = \tau_0(F) \text{ asa } \tau_0(G).$$

Observe that if expressions are restricted to formulae, then the range of ϕ_n and τ_n is the set of terms.

Example 3.2. Consider the expression

$$E = \text{first}(X + \text{next } X) \text{ fby next}(\text{next } Y + \text{first } Z + 3).$$

Then

$$\begin{aligned}\tau_0(E) &= \phi_0(\mathsf{first}\,(X + \mathsf{next}\,X))\ \mathsf{fby}\ \tau_0(\mathsf{next}\,(\mathsf{next}\,Y + \mathsf{first}\,Z + 3))\\ &= \phi_0(X + \mathsf{next}\,X)\ \mathsf{fby}\ \tau_1(\mathsf{next}\,Y + \mathsf{first}\,Z + 3)\\ &= X{:}0 + \phi_1(X)\ \mathsf{fby}\ \tau_2(Y) + \phi_0(Z) + 3\\ &= X{:}0 + X{:}1\ \mathsf{fby}\ Y\langle 2\rangle + Z{:}0 + 3.\end{aligned}$$

The following theorem formally states the effect of ϕ_n and τ_n on the interpretation of expressions.

Theorem 3.1. Let E be an expression, $\|$ an interpretation, and t in N. Then

$$\begin{aligned}(|\phi_n(E)|)_t &= (|E|)_n,\\ (|\tau_n(E)|)_t &= (|E|)_{t+n}.\end{aligned}$$

Proof (by induction on the structure of E).
(a) If E is a variable name, the theorem is obvious.
(b) Let ω be an r-ary standard operator, $F^{(1)} \ldots F^{(r)}$ expressions, and assume $E = (\omega, F^{(1)}, \ldots, F^{(r)})$. Then

$$\begin{aligned}(|\phi_n(E)|)_t &= (|(\omega, \phi_n(F^{(1)}), \ldots, \phi_n(F^{(r)}))|)_t\\ &= (|\omega|, (|F^{(1)}|)_n, \ldots, (|F^{(r)}|)_n) \quad \text{(ind. hyp.)}\\ &= (|(\omega, F^{(1)}, \ldots, F^{(r)})|)_n\end{aligned}$$

and analogously for τ_n.
(c) Let $E = \mathsf{first}\,F$, F some expression, then

$$\begin{aligned}(|\phi_n(E)|)_t &= (|\phi_0(F)|)_t\\ &= (|F|)_0 \quad \text{(ind. hyp.)}\\ &= (|E|)_n \quad \text{(def. of first)}\end{aligned}$$

and analogously for τ_n. With $E = \mathsf{next}\,F$ the argument is equally straightforward.
(d) Let $E = F\ \mathsf{fby}\ G$, F and G expressions, then

$$\begin{aligned}(|\phi_0(E)|)_t &= (|\phi_0(F)|)_t = (|F|)_0 = (|E|)_0,\\ (|\tau_0(E)|)_t &= (|\phi_0(F)\ \mathsf{fby}\ \tau_0(G)|)_t.\end{aligned}$$

Two cases arise,

(1) $t = 0$, then

$$(|\phi_0(F)\ \mathsf{fby}\ \tau_0(G)|)_t = (|\phi_0(F)|)_t = (|F|)_0 = (|E|)_{t+0},$$

(2) $t > 0$, then

$$\begin{aligned}(|\phi_0(F)\ \mathsf{fby}\ \tau_0(G)|)_t &= (|\tau_0(G)|)_{t-1} \quad \text{(def. of fby)}\\ &= (|G|)_{t-1} \quad \text{(ind. hyp.)}\\ &= (|E|)_t \quad \text{(def. of fby).}\end{aligned}$$

The argument for ϕ_m and τ_m, $m>0$, is easy and left to the reader.

(e) Let $E=F$ asa G, finally, F and G expressions. Then the theorem is a simple consequence of the definition of the asa operator.

■

Intuitively, ϕ_n and τ_n have the same effect on expressions as $:n$ and $\langle n\rangle$ have on variables. Yet they are not mere extensions as they, at the same time, effect a removal of all operators first and next.

The notion of *transformed program* is introduced. The transformed program of a given Lucid program P is an infinite equivalent one containing only extended terms.

Definition. Let P be a Lucid program. Then the *transformed program* P' of P is as follows:

(i) For every statement $X=E$ in P where E is a formula or is F asa G, F and G formulae, P' contains statements

$$X{:}\ i=\phi_i(E); \qquad X\langle i\rangle=\tau_i(E);$$

for all $i\geqq 0$.

(ii) For every statement $X=E$ fby F, E and F formulae, in P, P' contains

$$X{:}0 \quad =\phi_0(E); \qquad X\langle 0\rangle \quad =\phi_0(E) \text{ fby } X\langle 1\rangle;$$
$$X{:}i+1=\phi_i(F); \qquad X\langle i+1\rangle=\phi_i(F);$$

for all $i\geqq 0$.

(iii) P' contains no other statemnts.

Theorem 3.2. P and P' have the same solutions, and therefore the same minimal solution.

Proof. Immediate from the construction of P' and Theorem 3.1.

Since only the result value is of ultimate interest, usually a subprogram of P' is sufficient to define it. The *modified program* P_0 of P is the smallest set of statements in P' containing the definition of *result*:0 and being closed with respect to the reference structure, i.e. such that each name referenced in P_0 is also defined in it.

It should be evident that since P_0 is closed with respect to the reference structure, all solutions to P_0 are restrictions of corresponding solutions of P', hence also of P.

The dependency graph restrictions can be shown to imply that the modified program is finite. Intuitively speaking, this ensures that, at compile time, the total amount of storage necessary to evaluate P_0 iteratively can be predicted.

It is convenient to consider $X{:}i$ and $X\langle i\rangle$ as new variables neglecting that $:i$ and $\langle i\rangle$ have been interpreted as operators. When doing so, in order to avoid confusion, we shall denote these new variables by $X{\cdot}i$ and $X[i]$, respectively.

Definition. Given a variable name X, $X\cdot i$ and $X[i]$, $i\geqq 0$, are *qualified names* derived from X. Furthermore, $X\cdot i$ and $X[i]$ name the terms $X{:}i$ and $X\langle i\rangle$, respectively.

Definition. Let $X=G$ be a definition of X in the program P, E and F some formulae. The *defining term* of a qualified name derived from X is as follows. If $G=E$, or $G=E$ asa F, then

$$\text{def}(X\cdot i) \quad =\bar{\phi}_i(G), \qquad \text{def}(X[i]) \quad =\bar{\tau}_i(G),$$

and if $G=E$ fby F, then

$$\text{def}(X\cdot 0) \quad =\bar{\phi}_0(E), \qquad \text{def}(X[0]) \quad =\bar{\phi}_0(E),$$
$$\text{def}(X\cdot i+1)=\bar{\phi}_i(F), \qquad \text{def}(X[i+1])=\bar{\tau}_i(F),$$

for all $i\geqq 0$, and where the $\bar{\phi}_i$ and $\bar{\tau}_i$ differ from ϕ_i and τ_i only in that $\bar{\phi}_i(Y)=Y\cdot i$ and $\bar{\tau}_i(Y)=Y[i]$, for variable names Y.

Observe the close correspondence between defining terms and the statement transformation defined earlier. The only deviation, the defining term of $X[0]$ where $X=E$ fby F, is for a technical reason which will become clear subsequently.

Definition. A qualified name α *directly depends* on another qualified name β if β is an operand of $\text{def}(\alpha)$. The transitive closure of "directly depends" is the relation *depends*, and is denoted by $\alpha \gtrdot \beta$.

The following relates the edgelabelling conventions of the dependency graph of Section 2 to dependencies among qualified names. The theorem is used in Section 4 to show termination of the algorithms.

Theorem 3.3. Let α be $X[i]$ or $X\cdot i$, and β be $Y[j]$ or $Y\cdot j$. If $\alpha \gtrdot \beta$, then there is a path from X to Y in the dependency graph with a sum s of edge labels such that $s\geqq j-i$.

Proof. That there is a path from X to Y is obvious from the construction of the graph. The proof of $s\geqq j-i$ proceeds by induction on the path length k:

Basis. $k=1$: α directly depends on β.

(a) $X=$ first Y, hence $s=0$. Obviously $j=0$ and $i\geqq 0$.

(b) $X=$ next Y, hence $s=1$ and $j=i+1$.

(c) $X=f(Y)$, where f stands for a pointwise expression with Y as operand.

Therefore, $s=0$ and $i=j$. The other cases are verified as easily.

Step. k to $k+1$:

Assume the theorem holds for a path of length k or less, and that $\alpha \gtrdot \beta$ with a path of length $k+1$. There must be some Z on the path from X to Y, so there is some $\gamma=Z[r]$ or $Z\cdot r$ such that $\alpha \gtrdot \gamma$ and $\gamma \gtrdot \beta$. By hypothesis, $r-i\leqq s_1$ and $j-r\leqq s_2$ for the edge label sums s_1 and s_2 of the paths from X to Z and from Z to Y, and since $s=s_1+s_2$, the theorem follows.

■

Corollary 3.4. If the program P is in the subset, then its modified program P_0 is finite.

Proof. Straight-forward.

We will now derive a basic approach to compiling Lucid programs. Given a program, subsets of defined variables may be isolated which are self-contained with respect to their dependency structure, i.e. each variable definition in the subset is made in terms of other variables also defined in the subset. Such subsets are called *nests*.

If R is defined by $R = E$ asa F, then we may evaluate it as follows: Evaluate in a loop $|E|_0$, $|F|_0$; $|E|_1$, $|F|_1$; ... until $|F|_s$ is **true** for the first time. Then break out of the loop with $|E|_s$ as value of $|R|_t$, for all t. For the evaluation of E and F exactly those varibles forming the smallest nest containing R have to be evaluated.

Because of the subset constraints, the dependencies among all names defined by an asa expression can be recorded by a directed acyclic graph. Those variables which are the leaves of this graph are called *simple goals*, and we consider their compilation first.

In evaluating a simple goal, only variables defined by formulae and fby expressions need to be evaluated. Let ρ be any solution of the program P, and denote by $|X|_\rho$, $|X{:}i|_\rho$ and $|X\langle i\rangle|_\rho$ the values of X, $X{:}i$ and $X\langle i\rangle$, respectively. $(|\ldots|_\rho)_t$ is to denote the t-component of the value $|\ldots|_\rho, t$ in N. Let h be the homomorphism which maps expressions over qualified names into extended terms by replacing each qualified name by the term it names. Interpret expressions over qualified names not containing fby and asa as ordinary scalar expressions. We show that then an assignment to a qualified name α of the form $\alpha \leftarrow \mathrm{def}(\alpha)$ is consistent with the solution ρ in the following sense.

Theorem 3.5. Let α be a qualified name derived from X in P where $X = E$ or $X = E$ fby F, E and F formulae. Let ρ be a solution of P and t in N. Then the following is true: If the value of every qualified name β in $\mathrm{def}(\alpha)$ is $v(\beta) = (|h(\beta)|_\rho)_t$, then $v(\mathrm{def}(\alpha)) = (|h(\alpha)|_\rho)_t$, except when $\alpha = X[0]$ and $X = E$ fby F, in which case the theorem holds only for $t = 0$.

Proof. Follows directly from the definition of $:i$ and $\langle i\rangle$ and $\mathrm{def}(\alpha)$ in conjunction with Theorem 3.1 and 3.2. The exception for $X[0]$ has to be made because of the definition of its defining term. ■

This suggests the following approach to the compilation of simple goals. Collect all qualified names on which $R \cdot 0$ depends. Try to sort them in accordance with their dependencies. If they can be linearized, a straight-line program with statements $\alpha \leftarrow \mathrm{def}(\alpha)$ should compute initial components of the minimal solution σ of P.

Definition. Let Q be a list of qualified names α. Q is *admissible* if, for every α in Q, the defining term of α is a term (i.e. does not contain any fby or asa), and whenever $\alpha \gg \beta$, then β is in Q and precedes α.

Corollary 3.6. Let $Q = \langle \alpha_1, \ldots, \alpha_r \rangle$ be an admissible list of qualified names, σ the minimal solution of the program P. After the straight-line program

$$\alpha_1 \leftarrow \mathrm{def}(\alpha_1)$$
$$\vdots$$
$$\alpha_r \leftarrow \mathrm{def}(\alpha_r)$$

has been executed, the values of the α_i are

$$v(\alpha_i) = (|h(\alpha_i)|_\sigma)_0.$$

Proof (by induction on r).

Basis. $r=1$: Since Q is admissible, there is no qualified name on which α_1 depends, hence, by Theorem 3.5, $v(\text{def}(\alpha_1)) = (|h(\alpha_1)|_\rho)_0$, for any solution ρ of P, hence, in particular, for $\rho = \sigma$.

Step. r to $r+1$: Straight-forward from Theorem 3.5 and the induction hypothesis.

■

In order to obtain subsequent components of the solution σ observe the following: For every quiescent sequence X, any qualified name derived from X must be constant-valued as t varies, so it does not have to be recomputed. Names of the form $X[0]$ with $X = E \text{ fby } F$ obtain new values by computing $X[1]$ and assigning $X[0] \leftarrow X[1]$.

On basis of these observations, we should like to find an admissible list S of all qualified names on which $R \cdot 0$ depends and which can be split into lists S' and S'' with the following properties. S' contans all qualified names $X[0]$ where $X = E \text{ fby } F$, and all constant-valued qualified names. In order to iterate the evaluation of qualified names in S'', we append to the code for S'' statements of the form $X[i] \leftarrow X[i+1]$, where $X[i]$ is a name not in S'' but such that $\alpha > X[i]$ for some α in S'', and $X[i+1]$ occurs in S''. The program

```
        code for S'
loop:   code for S''
        appended statements
        goto loop
```

should then correctly evaluate subsequent components of the minimal solution. Section 4 will make these remarks precise.

4. Compilation Algorithms

We now develop algorithms for handling simple goals. A running example illustrates the compilation. As target language a bastard ALGOL was chosen for readability. The algorithms must be considered central, since only small modifications extend them to handle the full subset.

The code to be compiled is structured to consist of a prelude, essentially a straight-line program containing computations which have to be performed just once, e.g., of names of the form $X \cdot i$, followed by a loop which iterates the evaluation of nonquiescent expressions.

Since fby expressions are evaluated by computing a special case once, i.e. $\text{def}(X[0])$, and then iterating a general case, i.e. $\text{def}((X[1])$, and because of their possible interaction, the loop may have to be 'unrolled' a number of times,

thereby accomodating all special case computations in the enlarged prelude. This is done as follows.

Given a simple goal $R = E$ asa F, Algorithm A 4.1 constructs an operand list containing all qualified names needed to evaluate $E[0]$ and $F[0]$. If any name in this list is initially to be computed from the first part of a fby expression, the loop is unrolled, and a new list constructed for $E[1]$ and $F[1]$. This process is repeated until, in the last list L_n, all qualified names defined by a fby expression are computed from the general case. Also, all existing dependencies are recorded.

Algorithm A 4.2 linearizes the dependency graph constructed by A 4.1, subject to additional constraints, thereby determining a sequence in which to do the various computations. Subset restrictions imply that this is always possible. Algorithm A 4.3 determines the exact loop limits, and Algorithm A 4.4 generates actual (unoptimized) code.

Definition. A set of variables $X^{(1)} \ldots X^{(r)}$ is a *nest*, if the definition of each $X^{(i)}$ references only variables $X^{(j)}$ in the set.

Definition. A variable X is a *goal*, if X is defined by an asa expression. X is a *simple goal*, if it is a goal and the smallest nest containing X does not reference any goal.

Example 4.1. Consider the following segment of a Lucid program:

$R = E$ asa F; $\quad X = 0$ fby $X + 1$;
$E = X$; $\quad Y = 1$ fby $Y + Z$;
$F = Y$ gt 25; $\quad Z = 2$ fby $Z + Z$;

In it, R is a simple goal, and $\{R, E, F, X, Y, Z\}$ is the smallest nest containing R. The compilation of R illustrates the algorithms of this section.

Algorithm A 4.1.

Input: A simple goal $R = E$ asa F

Output: Operand lists $L_0, \ldots, L_n$ containing qualified names needed to evaluate R correctly. Also, for each name, a dependency list is constructed.

1. [Initialize]
   Set n to 0.
2. [Initialize next operand list]
   Set L_n to be the list $\langle E[n], F[n] \rangle$.
3. [*Process and extend* L_n. Construct dependency lists]
   Take next item α in L_n and initialize its dependency list D to be empty.

   For each qualified name β in def(α) do the following:
   (i) If β is not in $L_0, \ldots, L_n$, then append it to L_n.
   (ii) If β is not in D, then append β to D.

4. [Check if L_n is complete]
   If some items in L_n have not yet been processed by the previous step, then go to Step 3.

5. [Check if another list becomes necessary]
   If there is an item $X[0]$ in L_n where X is defined by a fby expression, then increment n by 1 and go to Step 2.
   Otherwise stop.

□

Example 4.2. Given the simple goal R of Example 4.1, A4.1 has the following output:

L_0:

items	dependencies
$E[0]$	$X[0]$
$F[0]$	$Y[0]$
$X[0]$	–
$Y[0]$	–

L_1:

items	dependencies
$E[1]$	$X[1]$
$F[1]$	$Y[1]$
$X[1]$	$X[0]$
$Y[1]$	$Y[0], Z[0]$
$Z[0]$	–

L_2:

items	dependencies
$E[2]$	$X[2]$
$F[2]$	$Y[2]$
$X[2]$	$X[1]$
$Y[2]$	$Y[1], Z[1]$
$Z[1]$	$Z[0]$

Note that L_2 does not contain any name $P[0]$, where $P = G$ fby H.

Theorem 4.1. For simple goals in subset programs Algorithm A4.1 halts.

Proof. Observe first, that a new list L_{r+1} is constructed only when L_r contains a name $X[0]$ where X is defined by a fby expression. Hence the n of A4.1 cannot exceed $k+1$, where k is the total number of fby expressions occurring in the smallest nest containing R.

Assume next that a list L_r grows infinitely. Then, since there are only finitely many sequence names, there will be qualified names α and β such that $\alpha \gtrdot \beta$, $\alpha = X[i]$ or $X \cdot i$, $\beta = X[j]$ or $X \cdot j$, and $j > i$. By Theorem 3.3 there is a cycle in the dependency graph with edge label sum $s \geqq j - i > 0$, contrary to subset syntax. Hence each list L_r is finite.

■

Algorithm A4.2 (sort operand lists).

Input: Operand lists $L_0, \ldots, L_n$ from Algorithm A4.1.
Output: Sorted operand lists $S_0, \ldots, S_n$

1. [Initialize]
   Set m to 0.

2. [Termination Condition]
   If $m > n$ then stop;
   otherwise set S_m to be the empty list.

3. [Determine Admissible Items]
   Scan L_m marking all items as "admissible" if their dependency lists are empty.

4. [Selection]
   Of all admissible item select one by applying the following criteria in sequence. As soon as one is applicable go to Step 5.
   (4.1) If some $X \cdot j$ is admissible, select the first such $X \cdot j$.
   (4.2) Select the first admissible $X[j]$ such that there is no $X[i]$ in L_m where $i<j$.
5. [Sort]
   Remove the selected item from L_m and from all dependency lists and append it to S_m.
6. [Sort Completion Condition]
   If L_m is empty, increment m by 1 and go to Step 2;
   otherwise go to Step 3.

□

Example 4.3. The sorted lists for input $L_0, \ldots, L_2$ of Example 4.2 are:

$$S_0 = X[0], E[0], Y[0], F[0],$$
$$S_1 = X[1], E[1], Z[0], Y[1], F[1],$$
$$S_2 = X[2], E[2], Z[1], Y[2], F[2].$$

Note that the concatenation $S_1 \circ S_2 \circ S_3$ forms an admissible list.

Before proving that A4.2 halts, the notion of close dependency is introduced. A qualified name α *closely depends* on β ($\alpha *> \beta$), if α depends on β and in the direct dependency chain $\alpha = \alpha_1 -> \alpha_2 -> \cdots -> \alpha_k = \beta$ each name α_i is derived from a sequence X which is defined by a formula or a fby expression.

Lemma 4.2. If $X[i] *> Y[j]$, then $X[i+1] *> Y[j+1]$.

Proof. (By induction on the length of the dependency chain.)

Basis. $X[i]$ directly depends on $Y[j]$: The lemma is clear except when $i=0$ and $X = E$ fby F. In this case, observe that $X[0]$ depends directly only on names of the form $Y \cdot j$, hence this case does not arise.

Step. Split the dependency chain such that $X[i] *> Z[r] *> Y[j]$. The lemma then follows from the induction hypothesis.

■

We use this lemma for proving that Algorithm A4.2 halts for subset programs. Despite Theorem 3.3 this is not obvious, as it is conceivable that the additional sort criteria may fail to select any qualified name from a set of admissible ones.

Theorem 4.3. Algorithm A4.2 halts for subset programs.

Proof. Observe that none of the lists L_i is empty, and that no item in L_i can depend on an item in L_j, $j>i$. Thus it suffices to show that the sort succeeds for any list L_r, after lists $L_0 \ldots L_{r-1}$ have been sorted. Two cases arise.

(1) No items were found admissible. There must be a circular dependency and, using Theorem 3.3, we see that the subset syntax is violated.

(2) No admissible items qualified for selection in Step 4. Since items of the form $X \cdot j$ depend only on items of the same form, this situation must be due to a violation of (4.2) in the algorithm. Let $\alpha_1, \ldots, \alpha_m$ be all admissible items, therefore, and assume none can be selected. Without loss of generality, assume that $\alpha_{p+1}, \ldots, \alpha_m$ cannot be selected because of some items $\alpha_1, \ldots, \alpha_p$. Since the first p items do not block each other's selection, they are derived from p different names and there exist inadmissible items $\beta_1, \ldots, \beta_p$, where

$$\beta_j = X^{(j)}[i_j], \quad \alpha_j = X^{(j)}[i_j + t_j], \quad 1 \leqq j \leqq p, \quad t_j > 0.$$

Furthermore, since the $\alpha_{p+1}, \ldots, \alpha_m$ cannot be selected because of $\alpha_1 \ldots \alpha_p$,

$$\alpha_j = X^{(j)}[i_k + t_j], \quad p < j \leqq m,$$

where k is such that $X^{(j)} = X^{(k)}$, $t_j > t_k$, and $k \leqq p$. Since the $\beta_1 \ldots \beta_p$ are inadmissible, they depend on some $\alpha_1 \ldots \alpha_m$. Let $f(j)$ be the smallest k such that $\beta_j \gtrdot \alpha_k$, $1 \leqq j \leqq p$, $1 \leqq k \leqq m$, and observe that β_j closely depends on α_k. Define g by

$$g(j) = \begin{cases} f(j) & \text{if } f(j) \leqq p \\ k & \text{if } f(j) > p, \text{ with } k \text{ such that } X^{(k)} = X^{(f(j))} \text{ and } k \leqq p. \end{cases}$$

Because of the ordering of admissible items, g is well-defined. Consider the sequence 1, $g(1)$, $g^2(1), \ldots, g^p(1)$. Since $g(t) \leqq p$, at least one value $j \leqq p$ occurs twice, hence there is a k such that $g^k(j) = j$. Using this construction, we obtain, by repeated application of Lemma 4.2,

$$X^{(j)}[i_j] \ast\!\!> X^{(j)}[i_j + \bar{t}_1 + \cdots + \bar{t}_k], \quad \text{where } \bar{t}_i = t_{f(g^{(i-1)}(j))}.$$

In conjunction with Theorem 3.3 we have then a violation of the subset syntax. ■

The following Algorithm combines the sorted lists from A4.2 into an admissible list (schedule) for code generation. This involves marking places at which to test for loop termination, and determining the exact loop boundaries. A4.4 generates code from this schedule.

Algorithm A4.3 (construct schedule).

Input: Sorted lists $L_0, \ldots, L_n$ from A4.2
Output: Schedule S

1. [Construct first part of S]
   Let T_i be a list containing only the symbol $\#_i$. Set S to be the concatenation of $L_0, T_0, \ldots, L_{n-1}, T_{n-1}$.
2. [Determine loop]
   Let $L_n = \langle \alpha_1, \ldots, \alpha_m \rangle$. Scan L_n marking an item $X[i]$ as "in-loop" if $X[i+1]$ is not in L_n. Let p be the smallest j such that α_j is marked "in-loop". Split L_n into the lists

   $L' = \langle \alpha_1, \ldots, \alpha_{p-1} \rangle$ (empty if $p = 1$)
   $L'' = \langle \alpha_p, \ldots, \alpha_m \rangle$

3. [Complete schedule]
   Append to S the lists L, T', L', T_n, where T' is the list $\langle \% \rangle$. Then stop.

□

Algorithm A4.4 (code generation).

Input: Schedule S from A4.3
Output: Code evaluating the simple goal $R = E$ asa F

1. [Initialize iteration counter and code S]
   Emit "**begin** $t \leftarrow 0$;"
   Scan S and code each item α as prescribed by Step 2. Then goto Step 3.
2. [Code item α in S]
   If α is a qualified name, then emit "$\alpha \leftarrow \text{def}(\alpha)$;",
   If α is $\%$, then emit "**repeat forever begin**",
   If α is $\#_i$, then emit "**if** $F[i]$ **then begin** $R.0 \leftarrow E[i]$; **goto** L **end**;".
3. [Code iteration counter increment and window shifts]
   Emit "$t \leftarrow t+1$;"
   Scan all qualified names α following $\%$ in S and do the following:
   If a name $X[i]$ occurs in $\text{def}(\alpha)$ but is not in S following $\%$, then generate instructions

$$\begin{aligned} X[i] &\leftarrow X[i+1]; \\ &\vdots \\ X[j-1] &\leftarrow X[j]; \end{aligned}$$

   where j is the smallest $k > i$ such that $X[k]$ follows the $\%$ in S.
   Collect all instructions so generated, delete duplications, and emit them sorted by the index of their left part.
4. [Code loop end]
   Emit "**end**; L: **end**;", then stop.

□

Theorem 4.7 below justifies Step 3 of A4.4.

Example 4.4. Algorithm A4.3 generates the following schedule for the lists of Example 4.3: $\langle X[0], E[0], Y[0], F[0], \#_0, X[1], E[1], Z[0], Y[1], F[1], \#_1, \%, X[2], E[2], Z[1], Y[2], F[2], \#_2 \rangle$, from which the following code is generated:

begin $t \leftarrow 0$;
$X[0] \leftarrow 0$;
$E[0] \leftarrow X[0]$;
$Y[0] \leftarrow 1$;
$F[0] \leftarrow Y[0]$ **gt** 25;
if $F[0]$ **then begin** $R.0 \leftarrow E[0]$; **goto** L **end**;
$X[1] \leftarrow X[0]+1$;
$E[1] \leftarrow X[1]$;
$Z[0] \leftarrow 2$;

```
        Y[1] ← Y[0]+Z[0];
        F[1] ← Y[1] gt 25;
        if F[1] then begin R·0←E[1]; goto L end;
repeat forever begin
                        X[2]←X[1]+1;
                        E[2] ← X[2];
                        Z[1] ←Z[0]+Z[0];
                        Y[2] ← Y[1]+Z[1];
                        F[2] ← Y[2] gt 25;
                        if F[2] then begin R·0←E[1]; goto L end;
                            t←t+1;
                        Z[0] ←Z[1];
                        X[1]←X[2];
                        Y[1] ← Y[2];
                end;
L: end;
```

Note that the code is well-suited for conventional code optimization.
□

Because of Corollary 3.6, a correctness proof of the compiled code amounts to showing that the generated schedule is an admissible list of qualified names, and has certain additional properties which make it possible to correctly iterate the evaluation of sequences. These results are stated in Theorem 4.7 and are proved with aid of the following observations.

Lemma 4.4. Let $L_0, \ldots, L_n$ be the operand lists constructed by A4.1. If $X[i]$ is in L_r, then $X[i+1]$ is in one of the lists $L_0, \ldots, L_{r+1}$.

Proof. If $X[i]$ is in L_r, then $E[r]$ or $F[r]$ closely depend on it. ■

Corollary 4.5. Let $L_0, \ldots, L_n$ be the lists constructed by A4.1. If $X[i]$ is in L_r for some r, then there is an index i_0 such that $X[i_0]$ is in L_n, and for every $X[j]$ in $L_0, \ldots, L_{n-1}$ we have $i_0 > j$.

Proof. Evident.

The following lemma summarizes the effect of Algorithm A4.2.

Lemma 4.6. Let S_r be the r-th sorted list output by Algorithm A4.2. Then the following is true.

(1) Any name $X \cdot i$ in S_r precedes every name $Y[j]$ in S_r.
(2) If $X[i]$ and $X[j]$, $j > i$, are in S_r, then $X[i]$ precedes $X[j]$.
(3) Either $E[r]$ or $F[r]$ is the last element in S_r.

Proof. A name of the form $X \cdot i$ depends only on names of the same form. Hence, (1) and (2) are a consequence of Step 4 of the algorithm. (3) follows from the construction of the corresponding operand list L_r by A4.1.
■

Theorem 4.7. Let S be the schedule constructed by Algorithm A4.3 from the sorted lists $S_0, \ldots, S_n$, and denote by S'' that part of S which follows the symbol $\%_0$. Then the following is true.

(1) If α is a qualified name in S and $\alpha > \beta$, then β is in S and precedes α.

(2) If β is a qualified name in S other than $E[r]$ and $F[r]$, $r \leqq n$, then there is a qualified name α in S such that $\alpha > \beta$. Furthermore, if β is in S_r, then so is α.

(3) If α is a qualified name in S'' and $X[i]$ is in def(α) but not in S'', then there are qualified names $X[i+1], \ldots, X[j]$ in S such that $X[j]$ is in S'' and the $X[i+1], \ldots, X[j-1]$ all precede $X[j]$.

Proof. (1) and (2) are immediate from the construction of operand lists by A4.1 and because A4.2 is a topological sort. For (3), if $X[i]$ is not in S_n, then, by repeated application of Lemma 4.4, there are names $X[i+1], \ldots, X[k]$ in $S_0, \ldots, S_n$, such that $X[k]$ is in S_n, but the other names are not, and therefore precede $X[k]$ in S. Furthermore, there are names $X[k], \ldots, X[j]$ in S_n (possibly $k=j$) such that $X[j+1]$ is not in S_n. Then $X[j]$ must be in S''. By Lemma 4.6 the $X[k], \ldots, X[j-1]$ all precede $X[j]$ in S_n and hence in S. If, on the other hand, $X[i]$ is in S_n, then the second part of the above argument completes the proof.

■

The first two parts of Theorem 4.7 show that the schedule S contains exactly those names on which the $E[0], F[0], \ldots, E[n], F[n]$ depend as well as those names themselves. The sequence of these names is admissible in the sense of Section 3. Furthermore, there are no "look-ahead" computations in the sense that a name α is computed unless the current iteration has become necessary and requires it. It is possible to design simpler algorithms which do a certain amount of look-ahead computations, but then the domain of correctly translated source programs would be more restricted.

The third part of the theorem is used to prove that subsequent loop iterations correctly evaluate subsequent components of the solution to the source program.

Recall that the schedule S is the concatenation of the lists S_0, $T_0, \ldots, S_{n-1}$, T_{n-1}, S', $\langle \%_0 \rangle$, S'', T_n, where the $T_i = \langle \#_i \rangle$ and $S_n = S' \circ S''$. Consequently, the generated code is of the following structure:

```
begin t←0;
      P0; R0; ... Pn−1; Rn−1; P′;
      repeat forever begin
                           P″; Rn;
                           t←t+1;
                           Q;
                     end;
      L: end;
```

where P_i, R_i P' and P'' are the translations of S_i, T_i, S' and S'', respectively. Note that only P' and S' can be empty, and that the only transfers of control apart

from the loop are in the R_i, all of which are "**goto** L" instructions. In the following, the above symbols are used to denote the various parts of the object program and the schedule.

We prove that, after executing an assignment to a qualified name α in the compiled code, the new value $v(\alpha)$ of α is precisely $(|h(\alpha)|_\sigma)_r$, where r is the current value of the variable t in the object code, σ is the minimal solution of the original program, and h is the homomorphism of Section 3.

Theorem 4.8. Let r be the value of t before executing the j-th assignment statement ot the qualified name α in the object program. Then the new value of α after the assignment is $v(\alpha)=(|h(\alpha)|_\sigma)_r$.

Proof.

Part 1. The j-th assignment statement (to α) is executed for the first time. By Corollary 3.6, the theorem is true for all assignments except those in part Q of the object code. For assignments in Q observe that they are of the form $X[i]\leftarrow X[i+1]$, that t has the value 1, and that there has been exactly one assignment to $X[i+1]$ which, furthermore, is not in Q, hence was performed when $t=0$.

Part 2. The j-th assignment is executed for the n-th time, $n>1$. Because of the control structure of the program, the assignment is in parts P'', Q, or R_n.

Case a. The assignment is in P'', hence of the form $\alpha\leftarrow\mathrm{def}(\alpha)$. It is verified that the assignment satisfies the hypotheses of Theorem 3.5: Any name α in $\mathrm{def}(\alpha)$ is either of the form $Y\cdot j$, in which case it names a constant-valued term and has the correct value because of Part 1 of the proof, or it of the form $Y[j]$ in which case it was assigned last in Q or in P'' preceding this statement, and, by induction hypothesis, has the value $(|h(\beta)|_\sigma)_r$.

Case b. The assignment is in R_n. This case is evident.

Case c. The assignment is in Q. This case is argued as in Part 1 of the proof.
■

Corollary 4.9. The code compiled by Algorithms A4.1 through A4.4 correctly evaluates the simple goal $R=E$ **asa** F which was input to A4.1.

Proof. If there is no t such that $(|F|_\sigma)_t$ is **true** and $(|F|_\sigma)$ is **false** for all $s<t$, then $(|R|_\sigma)_r$ is undefined for all r, and the compiled code will loop forever.

If there is some such t, on the other hand, then the compiled code must terminate and correctness follows from the previous theorem.
■

Note that because of the **if** ... **then** ... **else** ... it is possible that some sequence components have been evaluated, which are not used in some particular iteration. Although this does not affect the correctness proof for simple goals, it may introduce partial correctness in the presence of several goals, as worked out next.

5. Compilation of Programs

In general, programs may contain more than one variable defined by an asa expression. A suitable modification of the algorithms is in order, so that they can be used to compile the more general case. We describe informally how this may be done.

From the dependency graph G of a program a *goal graph* G' is constructed as follows:

The nodes of G' are all those nodes X of G such that X is a goal and/or the name *result*. If X and Y are names of nodes in G' and there is a path from X to Y in G, then draw an edge from X to Y in G'.

The graph G' derived as above is the goal graph of the program. It is easy to see that, because of the labelling conventions of the dependency graph, the goal graph is acyclic.

The algorithm modifications are now as follows. Each goal not named *result* is compiled into a procedure evaluating it. This procedure is to be called when the value of the goal is referenced by some other computation. Because of quiescence, each such procedure needs to be evaluated at most once, since the computed value cannot change subsequently. Efficient code taking advantage of this property is easily generated. The code for evaluating *result* then acts as main program. Note, that procedures may call each other but that, because of the acyclicity of the goal graph, there is no recursion.

In the construction of the operand lists by Algorithm A4.1 this involves the following. Upon discovery of a dependency on a qualified name derived from a goal X, the name is changed to $X \cdot 0$, its dependencies are not analyzed any further; code generation will generate a call to the appropriate procedure when coding $X \cdot 0$, and X is recorded as a goal to be compiled. Beginning with the compilation of the name *result*, this method, in a top-down fashion, eventually compiles all goals which are needed to evaluate *result*. In this way, the compilation of programs can be reduced to the compilation of goals, one at a time, using the methods of Section 4.

Theorem 5.1. The code π compiled for a goal $R = E$ asa F in a subset program evaluates R correctly, provided the correct values of each goal Y referenced in π is available in $Y \cdot 0$.

Proof (informal). Since the goal graph is acyclic, π cannot reference $R \cdot 0$. If R is a simple goal, the theorem follows from Corollary 4.9. For arbitrary goals, an induction completes the proof.
■

Corollary 5.2 (partial compiler correctness). Let π be the program compiled for the source program P in the subset, and let σ be the solution of P. Then, if P halts, it prints

$$(|\text{output}|_\sigma)_0$$

and if $(|\text{output}|_\sigma)_0 = \perp$, then π does not halt.

Proof. If π halts, the corollary follows from the previous theorem. If the value of output is *undefined*, then there is at least one goal which has this value, and the corresponding loop in π will not terminate.
■

Thus only partial correctness has been accomplished in the presence of more than one **asa** expression. Recall that in the dependency analysis of Algorithm A4.1 no distinction is made between dependencies due to strict operators, such as, for example, the arithmetic operators, and dependencies due to the non-strict operator **if**–**then**–**else**–. As a consequence, this operator is strict in this implementation. So it is possible, that a component of a sequence is evaluated which, due to a particular value configuration of the particular iteration would not be required. When such a component happens to reference a goal whose value otherwise would not be needed, then partial correctness may result.

There are many situations in which this problem can be overcome by suitable modifications of our approach. Unfortunately, there is no easy syntactic definition of those situations, and a general solution appears intricate and difficult. On the object code level it would involve, roughly speaking, a demand-driven set of recursive co-routines.

6. Conclusions

As already indicated, the fundamental choice of this implementation has been to view sequences as streams to be evaluated iteratively. This choice also establishes the analogy between Lucid and data-flow languages. The subset restriction expressed by the constraints on the dependency graph of Section 2 ensures that, in effect, no value component of a given stream requires the value of a future component of the same stream. Thus it is possible to predict at compile time the total amount of storage needed to evaluate each stream, which is given by the number of qualified names generated. In light of this, and since a relaxation of this constraint appears to force an implementation to become more interpretive, the subset choice seems to be appropriate. It has not been proved whether the expressive power of the subset is strictly smaller than that of the full language, but such a proposition seems unlikely to hold.

Several other points emerge from this work. Primarily it is seen that it is possible to prove compilers correct for larger languages. Tools for this are available, and, in many cases, the complexity of the task is not prohibitive. Essential for this, however, is a formalized semantics of the source language which for many procedural languages is not available. Also, and this was one of the initial difficulties in compiling Lucid, for non-procedural languages it is often difficult to factor the language into orthogonal constructs given presently available machine architectures, so that both design and correctness proof of a compiler can be modularized. In this paper, the problem was solved to some extent once the key rôle of the **asa** operator was recognized.

Furthermore, it is felt that the style of the analysis performed by the compiler is applicable to a broader class of languages. The influence which different

machine environments, e.g. network machines, might have on this analysis, and how it can be broadened to effect optimizing source transformations, seems particularly promising for future research.

The features of the source language, finally, deserve attention also. Since Lucid is a formal proof system, the proof of a Lucid program is essentially a sequence of source transformations, which make increasingly more apparent the various properties implied by the original program. It is interesting also, that a language motivated by program proving shares essential properties with data-flow languages which are motivated by the study of parallelism. Thus the work should stimulate deeper insights.

Acknowledgements. Many stimulating discussions with E. Ashcroft, A. Blikle, and T. Maibaum greatly helped shape this paper. A number of unusual Lucid programs which T. Cargill wrote contributed to the original designs on which the experimental compiler is based. Thanks are also due to the referees whose suggestions were valuable for improving this work.

References

1. Allen, F.E.: Program optimization. In: Annl. review of autom. programming (Halpern, Shaw, eds.), Vol. 5, pp. 239–308. New York: Pergamon 1969
2. Ashcroft, E.A.: Program proving without tears. Proc. of the Intl. Symp. on Proving and Improving Programs, pp. 99–111, Senans, France, July 1975
3. Ashcroft, E.A., Wadge, W.: Lucid, a non-procedural language with iteration. Comm. ACM **20**, 7, 519–526 (1977)
4. Ashcroft, E.A., Wadge, W.: Lucid, a formal system for writing and proving programs. SIAM J. Comput. **5**, 336–354 (1976)
5. Ashcroft, E.A., Wadge, W.: Lucid, scope structures and defined functions. Tech. Rept. CS-76-22, Dept. of Comp. Sci., University of Waterloo, Nov. 1976, 28 pp.
6. Cargill, T.A.: Deterministic operational semantics for Lucid. Tech. Rept. CS-76-19, Dept. of Comp. Sci., University of Waterloo, June 1976, 35 pp.
7. Chirica, L.M., Martin, D.F.: An approach to compiler correctness. Intl. Conf. on Reliable Software, June 1976, pp. 96–103
8. Dennis, J.B.: First version of a data-flow language. Proj. MAC Memo 61, MIT, May 1975
9. Floyd, R.W.: Assigning meaning to programs. In: Math. aspects of comp. sci., Vol. 19, pp. 19–32. Providence, R.I.:
10. Guttag, J.: Abstract data types and the development of data structures. Suppl. of Proc. of ACM Conf. on Data, Salt Lake City, Utah, Mar. 1976, pp. 37–46
11. Hoare, C.A.R., Wirth, N.: An axiomatic definition of the programming language PASCAL. Acta Informat. **2**, 335–355 (1973)
12. Hoffmann, C.M.: Design and correctness proof of a compiler for Lucid. Tech. Rept. CS-76-20, Dept. of Comp. Sci., Univ. of Waterloo, May 1976, 85 pp.
13. Kosinski, P.R.: A data-flow programming language. IBM Res. Rept. RC-4264, Mar. 1973, 134 pp.
14. London, R.L.: Correctness of two compilers for a LISP subset. A.I.Memo 151, Stanford Univ., 1971
15. McCarthy, J., Painter, J.A.: Correctness of a compiler for arithmetic expressions. In: Math. aspects of comp. sci., Vol. 19. Providence, R.I.: 1967
16. Miller, R.E., Cocke, J.: Configurable computers: A new class of general purpose machines. IBM Res. Rept. RC-3897, June 1972, 14 pp.

17. Milner, R., Weyrauch, R.: Proving compiler correctness in a mechanized logic. Machine Intelligence, Vol. 7, pp. 51–71, Univ. of Edinburgh, 1973
18. Morris, F.L.: Advice on strucuring compilers and proving them correct. ACM Symp. on Principles of Progr. Lang. Boston, 1973, pp. 144–152
19. Rumbaugh, J.E.: A parallel asynchronous architecture for data-flow languages. MIT Proj. MAC Rept. TR-150, May 1975, 319 pp.
20. VanEmden, M.: Verification conditions as representation for programs. Res. Rept. CS-76-03, Dept. of Comp. Sci., Univ. of Waterloo, Jan. 1976, 21 pp.

Received May 12, 1976