Acta Informatica 13, 9–20 (1980)

Acta Informatica



# Semantic Properties of Lucid's Compute Clause and its Compilation*

Christoph M. Hoffmann

Computer Science Department, Purdue University, West-Lafayette, IN, USA

**Abstract.** Compilation algorithms for a subset of the programming language Lucid are extended so that programs with nested *compute* clauses can be compiled.

The extension is derived from the semantic properties of the *compute* clause construct, and is applicable to a broad class of compilation algorithms for a subset of Lucid. A correctness proof of the extension is also given.

## Introduction

Software design methodology has recognized the need for structuring large programs so as to reduce their complexity and increase their clarity, see, e.g., [4]. Similar attempts at structuring very large proofs have been proposed recently, e.g. [7, 10], and have been guided by structuring principles of programs.

We present a structured compiler design and a structured correctness proof. In contrast to the above cited work, our structuring is based on semantic properties of the source language compiled. This approach has also been advocated in [8].

The source language under consideration is Lucid [1, 2, 12, 13], a nonprocedural, proof-oriented programming language. We define the basic language language, in the sense of [1] and [2], and isolate a subset by excluding certain language elements. In [6] we have presented an algorithm $\mathscr{A}$ for compiling programs in this subset, and have given a correctness proof for $\mathscr{A}$. In [5] a different compilation algorithm can be found. We study here how to extend the algorithm to compile correctly the larger language allowing the compute clause construct of [2].

* This work has been supported in part by an NSF grant MCS 78-01812

0001-5903/80/0013/0009/$02.40

We note that such extensions of $\mathscr{A}$, and its correctness proof, are formulated without assuming detailed properties of $\mathscr{A}$. Consequently, we have accomplished a modularization of the compiler and of its proof. Furthermore, other compilation algorithms for subsets of Lucid may be correctly extended in the same way.

## Informal Description of Lucid

A Lucid program is a set of equations describing sequences of values assumed by individual variables (histories). For loop-structured algorithms, we distinguish the value of a variable $X$ at the first iteration (denoted by *first* $X$), the current value of $X$ (simply $X$), and the value of $X$ in the next iteration (*next* $X$). Consider, for example, the following program which computes the factorial of its input.

**Example 1.**

```
read(N);
C:=0;    F:=1;
while C<N do begin
      F:=F*C+F;
      C:=C+1;
      end;
print(F);
```

The history of $C$ in the computation of the algorithm may be defined by

*first* $C=0$
*next* $C=C+1$

hence is the sequence $\langle 0, 1, 2, \ldots\rangle$. The *fby* operator (followed by) is used to abbreviate the pair by the single equation

$C=0$ *fby* $C+1$.

Note that in this equation $C$ is more naturally interpreted as the full history of $C$ than $C$'s current value. Likewise, $F=1$ *fby* $F*C+F$ defines the history of $F$ as the sequence $\langle 1, 1, 2, 6, 24, \ldots\rangle$, while the history of $N$ is given by $N=$ *first* input. Note that $N$ does not change value throughout the computation, hence its history is a constant sequence.

There is a binary operator *asa* (as soon as) in Lucid, which is used to 'extract' values from a loop: Let $A=\langle a_0, a_1, a_2, \ldots\rangle$ be a sequence, and $B=\langle b_0, b_1, b_2, \ldots\rangle$ a sequence of boolean values, then the expression $A$ *asa* $B$ is the (constant) sequence $\langle a_t, a_t, \ldots\rangle$ where $b_t$ is **true** for the first time, i.e. $b_s=$ **false** for all $s<t$. If no such $t$ exists, then the expression denotes the sequence $\langle$**undefined, undefined,** $\ldots\rangle$.

**Example 2.** The algorithm of the previous example is in Lucid

$N=$ *first* input
$C=0$ *fby* $C+1$
$F=1$ *fby* $F*C+F$
result $=F$ *asa* not $(C<N)$

Note that the algorithm has been described without specifying flow of control or any assignments, and that the order in which the equations are written is unimportant. It should be added that this equational specification method can express a richer set of algorithms than loop-structured ones, but it cannot achieve a nesting of loops without additional constructs.

Lucid has a construct which, loosely speaking, nests loops. Consider the following Lucid program

**Example 3.**

```
N = 1 fby N * 2
compute R using N
    C = 0 fby C + 1
    F = 1 fby F * C + F
    result = F asa not (C < N)
end
result = R.
```

The ALGOL equivalent of this program is

```
N := 1;
while true do begin
    C := 0; F := 1;
    while C < N do begin
        F := F * C + F;
        C := C + 1
        end;
    R := F; N := N * 2;
    print (R);
    end;
```

The 'compute...end' brackets a program segment almost identical to the program of Example 2. Informally, we have nested the computation of the first program which is to compute the factorial for every component of the sequence $N$, now globally referenced, and the resulting values become the sequence $R$. The containing program segment may be visualized by

```
N = 1 fby N * 2
R = N factorial
result = R
```

and is to output the sequence $\langle 1!, 2!, 4!, \ldots \rangle$.

Since, for every activation of the nested clause, $C$ and $F$ have an entire sequence of values as history, the history of these variables in the context to the program is a sequence each of whose components in turn is a sequence recording the values assumed during an activation of the nested clause. Such sequences of sequences are indexed by two parameters, $t_0$ and $t_1$, where $t_0$ corresponds to the inner and $t_1$ to the outer loop. Hence $C_{t_0 t_1}$ is the value of $C$ during the $t_1$ activation of the

nested clause in the $t_0$ iteration of the nested loop. Formal precision is obtained by interpreting every variable as a sequence of infinite dimensionality (this avoids conflicts of dimensionality), and viewing the compute clause as an operation on dimensions.

The syntax of the **compute clause** is as follows

compute ⟨variable⟩ using ⟨variable list⟩
⟨set of assertions⟩
end.

The clause defines the variable following 'compute', called the **subject** of the clause. The variables in the list following 'using' are the **global variables,** the set of assertions is the **body** of the clause. All variables in the body which are not global are **local variables.** The special variable **result** is always local and refers to the subject of the clause. The exact semantics of the clause is defined only indirectly by a syntax transformation given in the next section, and specific properties will have to be derived.

## Formal Definition of Lucid

We outline briefly the syntax and semantics of Lucid in the style of [1]. That is, we define what has been called **basic Lucid** in [12, 13]. The reason for this is that the later versions, which possess a variety of additional clause structures, have semantics which is defined indirectly by syntactic transformations into equivalent basic Lucid, whose semantics was defined in [1].

Without loss of generality, we fix a standard alphabet $\Sigma$, a set of variables $V$, and interpret $\Sigma$-terms in the usual sense by a **standard structure** $S$. The **range** of $S$ is denoted $U_S$, and contains at least the values **undefined, true, false,** and the integers. Given an indexing set $X$, $S$ is extended to the unique **power structure** $C$ by taking the crossproduct $S^X$, as in [1], Sect. 2.2. Note that the range of $C$ is the set $U_C$ of functions from $X$ into $U_S$. In the following, $X$ is $N^N$, i.e. the set of infinite sequences of natural numbers, except where indicated otherwise.

Let $G$ be the set of **operation symbols** in $\Sigma$, $F$ the set {*first*, *next*, *fby*, *asa*, *latest*, $latest^{-1}$} of **Lucid functions** (none of which are in $\Sigma$). Denote by $E$ the set of $\Sigma \cup F$-terms without quantifiers and '$=$', $E_0$ the subset of $E$ of terms without *latest* and $latest^{-1}$ as well.

The interpretation Comp($S$) of $(\Sigma \cup F)$-terms is now as follows, where $U_C$ is the set of functions from $N^N$ into $U_S$:

*Definition.* (Ashcroft and Wadge) If $S$ is a standard $\Sigma$-structure, then Comp($S$) is the unique $(\Sigma \cup F)$-structure $C$ which extends $(S^{N^N})^*$ to the larger alphabet as follows:

For $\alpha$, $\beta$ in $U_C$ and $\bar{t} = t_0 t_1 t_2 \ldots$ in $N^N$

(1) $(first_C(\alpha))_{\bar{t}} = \alpha_{0 t_1 t_2 \ldots}$

(2) $(next_C(\alpha))_i = \alpha_{i_0+1\, i_1 i_2 \ldots}$

(3) $(\alpha\ asa_C\ \beta)_i = \alpha_{s i_1 i_2 \ldots}$ if $\beta_{s i_1 i_2 \ldots}$ is **true**, and for all $r < s$, $\beta_{r i_1 i_2 \ldots}$ is **false**;
**undefined** if no such $s$ exists

(4) $(\alpha\ fby_C\ \beta)_i = \alpha_{0 i_1 i_2 \ldots}$ if $i_0 = 0$
$\beta_{i_0 - 1\, i_1 i_2 \ldots}$ otherwise

(5) For $\omega$ in $G$, $(\omega_C(\alpha, \beta, \ldots))_i = (\omega_S(\alpha_i, \beta_i, \ldots))$

(6) $(latest_C\ \alpha)_i = \alpha_{i_1 i_2 \ldots}$

(7) $(latest^{-1}{}_C\ \alpha)_i = \alpha_{0 i_0 i_1 i_2 \ldots}$

A **basic** Lucid **program** $P$ is a set of $(\Sigma \cup F)$-terms of the form $v = \phi_v$, where $\phi_v$ is in $E$, and such that every variable $v$ in $P$ is defined by at most one term, and the variable input is not defined. If, furthermore, every term $\phi_v$ is in $E_0$, i.e. does not contain *latest* and $latest^{-1}$, then $P$ is a **simple program.** [6] has dealt with the compilation of simple programs.

Given a fixed interpretation $\alpha$ of input, the **solution** ('**meaning**' in [1]) of $P$ is the minimal (least defined) $C$-interpretation $\sigma$ which satisfies $P$, i.e. $\models_{\sigma,\alpha} P$. Every basic Lucid program has a unique solution [1].

In order to enhance the clarity of Lucid programs, certain syntactic constructs have been introduced in [2]. We will study the nature of the compute clause, whose semantics is defined by a syntactic transformation changing programs with compute clauses into basic programs.

A compute clause is equivalent to the set of terms $v = \phi_v$ obtained by

(1) Renaming all local variables except result with new names not occurring elsewhere in the program,

(2) Replacing every global variable $X$ by *latest* $X$ in the body,

(3) Replacing 'result $= \phi$' by $Y = latest^{-1}(\phi)$, where $Y$ is the subject of the clause and deleting the **compute**...⟨variable list⟩ and **end.**

**Example 4.** The compute clause of Example 3 is equivalent to

$C = 0\ fby\ C + 1$
$F = 1\ fby\ F * C + F$
$R = latest^{-1}(F\ asa\ \text{not}\ (C < latest\ N))$

We can now define structured Lucid programs. A **basic assertion** is a $(\Sigma \cup F)$-term of the form $v = \phi_v$ where $\phi_v$ is a term in $E_0$. Then

⟨program⟩ ::= compute output ⟨globals⟩ ⟨clause body⟩
⟨globals⟩ ::= ⟨empty⟩
| using ⟨variable list⟩
⟨clause body⟩ ::= end
| ⟨assertion⟩ ⟨clause body⟩
⟨variable list⟩ ::= ⟨variable⟩
| ⟨variable⟩, ⟨variable list⟩
⟨assertion⟩ ::= ⟨basic assertion⟩
| ⟨clause⟩
⟨clause⟩ ::= compute ⟨variable⟩ ⟨globals⟩ ⟨clause body⟩

In addition, the following must hold. The ⟨globals⟩ of output is either empty or 'using input', every variable is defined at most once, either as the subject of a clause, or by a basic assertion, and the expression in each result definition is **syntactically quiescent** (see [1], Sect. 4.1) as defined below.

*Definition.* An expression $E$ is **syntactically quiescent** if

(a) $E$ is a constant, or

(b) $E$ is a global variable of a compute clause $C$ and $E$ is in the body of $C$, or

(c) $E=\omega(F_1, \ldots, F_r)$ where $\omega$ is in $G$, or

(d) $E=first\ F$, where $F$ is any expression, and the $F_i$ are quiescent, or

(e) $E=next\ F$, where $F$ is quiescent, or

(f) $E=F\ asa\ G$, for any expressions $F$ and $G$.

Note that quiescence differs from constancy in that it is a local property, due to (b).

A **simple program** is a program without any nested clauses. Simple programs may be interpreted in a simpler semantic structure, Loop($S$). For Loop($S$), the power structure $S^X$ is $S^N$, instead of $S^{N^N}$. The definition of Loop($S$) is the same as the definition of Comp($S$) with $N^N$ replaced by $N$, $\bar{t}$ by $t$, deleting $t_1 t_2 \ldots$, and parts (6) and (7) from the definition.

*Definition.* If $S$ is a standard $\Sigma$-structure, then Loop($S$) is the unique $(\Sigma \cup F)$-structure $C$ which extends $(S^N)^*$ to the larger alphabet as follows:

For $\alpha, \beta$ in $U_C$ and $t$ in $N$

(1) $(first_C\ \alpha)_t = \alpha_0$

(2) $(next_C\ \alpha)_t = \alpha_{t+1}$

(3) $(\alpha\ asa_C\ \beta)_t = \alpha_s$ if $\beta_s$ is **true** and $\beta_r$ is **false** for all $r<s$; **undefined** if no such $s$ exists.

(4) $(\alpha\ fby_C\ \beta)_t = \begin{cases} \alpha_0 & \text{if } t=0; \\ \beta_{t-1} & \text{otherwise.} \end{cases}$

(5) For $\omega$ in $G$,
$(\omega_C(\alpha, \beta, \ldots))_t = (\omega_S(\alpha_t, \beta_t, \ldots))$

A term is **interpretable in Loop($S$)** if it does not contain the Lucid functions *latest* and $latest^{-1}$. Simple programs are interpretable in Loop($S$) since they contain only terms interpretable in Loop($S$). In the following we show that the Loop($S$) interpretation of such programs is essentially the same as their Comp($S$) interpretation.

## Properties of the Compute Clause

The purpose of this section is to derive an abstraction principle which will permit us to replace an entire compute clause $B$ with subject $R$ and global variables $Y^{(1)}, \ldots, Y^{(r)}$ by an equation

$$R = F(Y^{(1)}, \ldots, Y^{(r)})$$

where $F$ denotes a pointwise operation, that is, for some function $f$: $U_S^r \to U_S$,

$$|F(Y^{(1)}, \ldots, Y^{(r)})|_{t_0 t_1 \ldots} = f(|Y^{(1)}|_{t_0 t_1 \ldots}, \ldots, |Y^{(r)}|_{t_0 t_1 \ldots})$$

The significance of this result, for the purpose of extending a subset compiler, is as follows.

Recall that programs with nested compute clauses have to be interpreted in Comp($S$) due to the implicit presence of the Lucid functions *latest* and *latest*$^{-1}$. However, intuitively speaking, after replacing all global variables in the body of an innermost clause by suitable constants, we may interpret the clause in Loop($S$) to obtain a correct component value for its subject. Inductively applied, this will provide a compiling strategy for such programs which amounts to extending a subset compiler by suitable calling sequences of subroutines, in the compiled code, each evaluating one compute clause.

**Lemma 1.** (Ashcroft and Wadge). *For any standard structure $S$ and any term $\phi$ and set of terms $\Gamma$ all interpretable in* Loop($S$)

$$\Gamma \models_{\text{Comp}(S)} \phi \quad \text{iff } \Gamma \models_{\text{Loop}(S)} \phi$$

*Proof.* [1], Theorem 1.

**Lemma 2.** *Let $B$ be a compute clause with global variables $Y^{(1)}, \ldots, Y^{(r)}$ and not containing nested compute clauses, and assume given functions $f_j$,*

$$f_j\colon\ U_S^r \to [N \to U_S]$$

*which, for every interpretation $\alpha_i$ in $U_S$ of the $Y^{(i)}$, yield the interpretation of the local variables $X^{(1)}, \ldots, X^{(n)}$ for the minimal solution of $B$ in* Loop($S$). *Then, for every interpretation $\alpha_i'$: $N^N \to U_S$ of the $Y^{(i)}$, the interpretation of the local variable $X^{(j)}$ for the minimal solution of $B$ in* Comp($S$) *is*

$$|X^{(j)}|_{t_0 t_1 t_2 \ldots} = f_j(\alpha_1'(t_1 t_2 \ldots), \ldots, \alpha_r'(t_1 t_2 \ldots))(t_0).$$

*Proof.* Consider the minimal solution for $B$ in Loop($S$) after interpreting each global variable $Y^{(i)}$ as a constant $\alpha_i$ in $U_S$. This induces, for each local variable $X^{(j)}$ of $B$, a function

$$f_j\colon\ U_S^r \to [N \to U_S]$$

which yields the interpretation of $X^{(j)}$ consistent with the solution:

$$|X^{(j)}|_{t_0} = f_j(\alpha_1, \ldots, \alpha_r)(t_0).$$

Let $\alpha_i'$: $N^N \to U_S$ be an interpretation of $Y^{(i)}$, and consider the minimal solution of $B$ in Comp($S$) given $\alpha_1', \ldots, \alpha_r'$. Because $Y^{(i)}$ denotes, inside $B$, the term *latest* $Y^{(i)}$, we have, inside $B$,

$$|\mathit{latest}\ Y^{(i)}|_{t_0 t_1 t_2 \ldots} = |Y^{(i)}|_{t_1 t_2 \ldots} = \alpha_i'(t_1 t_2 \ldots)$$

By Lemma 1, therefore, the interpretation of $X^{(j)}$ consistent with the minimal solution of $B$ in Comp($S$) is

$$|X^{(i)}|_{t_0 t_1 t_2 \ldots} = f_j(\alpha_1'(t_1 t_2 \ldots), \ldots, \alpha_r'(t_1 t_2 \ldots))(t_0). \quad \blacksquare$$

The lemma permits us to characterize the semantics of the compute clause without appealing to syntax transformations.

**Theorem 1.** Let $B$ be a compute clause with global variables $Y^{(1)}, \ldots, Y^{(r)}$ and subject $R$, and assume given a function $f_r: U_S^r \to U_S$ which, for every interpretation $\alpha_i$ in $U_S$ of the $Y^{(i)}$, yields the interpretation of the **result** of $B$ for the minimal solution of $B$ in Loop($S$). Then, for every interpretation $\alpha_i': N^N \to U_S$ of the $Y^{(i)}$, the interpretation of $R$ for the minimal solution of $B$ in Comp($S$) is

$$|R|_{t_0 t_1 \ldots} = f_r(\alpha_1'(t_0 t_1 \ldots), \ldots, \alpha_r'(t_0 t_1 \ldots)).$$

*Proof.* (By induction on the nesting structure of clauses).
**Basis.** $B$ does not contain nested compute clauses. By Lemma 2 and the quiescence of the result expression, therefore, the interpretation of **result** consistent with the minimal solution of $B$ in Comp($S$), given the $\alpha_i'$, is

$$\begin{aligned} |\mathbf{result}|_{t_0 t_1 t_2 \ldots} &= |\mathbf{result}|_{0 t_1 t_2 \ldots} \quad \text{(quiescence)} \\ &= f_r(\alpha_1'(t_1 t_2 \ldots), \ldots, \alpha_r'(t_1 t_2 \ldots)). \end{aligned}$$

Since, by the definition of the compute clause,

$$R = latest^{-1}(\mathbf{result})$$

we have

$$\begin{aligned} |R|_{t_1 t_2 \ldots} &= |latest^{-1}(\mathbf{result})|_{t_1 t_2 \ldots} \\ &= |\mathbf{result}|_{0 t_1 t_2 \ldots} \\ &= f_r(\alpha_1'(t_1 t_2 \ldots), \ldots, \alpha_r'(t_1 t_2 \ldots)). \end{aligned}$$

**Induction Step.** Replace every directly nested clause $B_i$ with subject $R$ and global variables $Z^{(1)} \ldots Z^{(m)}$ by an equation $R = F(Z^{(1)}, \ldots, Z^{(m)})$, where $F$ is the pointwise extension of a function $f_r': U_S^m \to U_S$. Now the theorem follows from the argument of the induction basis.

The theorem has an interesting corollary:

**Corollary 1.** *Let $B$ be a compute clause all of whose global variables are quiescent. Then the subject of $B$ is quiescent in the containing clause.*

The corollary can be used to define an optimizing transformation which reduces the nesting level of such clauses in the compiled code. Such optimization corresponds to moving invariant computations out of loops in the compilation of procedural languages.

## Extension of Compilation Algorithms

Based on the semantic properties of the compute clause previously derived, we now develop a uniform method for extending a compilation algorithm $\mathcal{A}$ which compiles correctly a subclass of simple programs. Correctness is defined in the sense of Theorem 4.8 of [6], that is, if, for a certain value $t$ in $N$, the value of a

variable $X$ at $t$ is required (in order to compute the value of output), then the compiled code computes this value as $\alpha(t)$, where $\alpha$ is the interpretation of $X$ for the minimal solution of the program in Loop($S$).

The subclass which can be compiled correctly is defined by a set $\mathscr{R}$ of syntactic restrictions. In the following we also denote by $\mathscr{R}$ the induced decision procedure checking whether a program is in the subset. Depending on the sophistication of $\mathscr{A}$, $\mathscr{R}$ will be more or less restrictive. See [6] and [5] for possible definitions of $\mathscr{R}$.

Our extension will yield an algorithm $\mathscr{A}$ which compiles programs with nested compute clauses. Intuitively, we will associate with each compute clause a simple program. Since, by Theorem 1, the clause may be treated in the containing clause as a certain equation, this association is conceptually simple. Using $\mathscr{A}$, we compile each such simple program separately into a procedure. The global variables of the clause will then become the parameters of these procedures. Certain complications which arise are accommodated by suitable calling mechanisms.

*Definition.* An operator $f$ is **unsafe** if, for variables $X$, $Y$, ... such that their interpretation functions cannot yield **undefined**, and for some $\bar{t}$ in $N^N$, $|f(X, Y, \ldots)|_{\bar{t}}$ is **undefined.**

The *asa* operator is unsafe. Observe that due to the combination of the *asa* and the nonstrict **if-then-else** there are simple programs for which variables have to be evaluated on demand only, i.e., by the **delayed evaluation** rule of Vuillemin [9]. Thus, loosely speaking, an unsafe operation should not be evaluated in the compiled code unless a particular value configuration arises which requires this.

Since we assumed correctness of $\mathscr{A}$, there are two cases which characterize the algorithm:

(1) $\mathscr{A}$ implements a procedure which, for argument $t$ in $N$ and variable name $X$ in the program computes $\alpha(t)$, where $\alpha$ is the interpretation of $X$ for the minimal solution of the program. That is, $\mathscr{A}$ implements delayed evaluation.

(2) For all programs satisfying $\mathscr{R}$ we can show that the evaluation of a variable $X$ at $t$, whether required or not, cannot lead to a nonterminating computation.

We assume in the following, that Case (1) is given.

We begin with defining a class of programs compilable by the extension of $\mathscr{A}$. Consider the restriction $\mathscr{R}$: Viewed algorithmically, $\mathscr{R}$ analyzes syntactically a set of terms $v=\phi_v$, where $\phi_v$ is in $E_0$. Given a program $\mathscr{P}$, associate with every compute clause $B_i$ of $\mathscr{P}$ a set $P_i$ if terms of the above form as follows:

Replace every compute clause $B_j$ with subject $X$ and global variables $Y^{(i)}$ directly nested in $B_i$ by the term $X=f_X(\ldots Y^{(i)}\ldots)$, where $f_X$ is nonstrict, unsafe and pointwise. Furthermore, for every global variable $Z$ of $B_i$ add a term $Z=i_Z(z)$, where $z$ is a symbolic constant and $i_Z$ a partial identity, i.e. pointwise and unsafe. Let $P_i$ be the so transformed body of $B_i$. $P_i$ is called the **associated simple program** of $B_i$. Note that adding the terms for the global variables captures their quiescence and the possibility, that an unwarrented evaluation of $Y$ may lead to nontermination.

*Definition.* If every associated simple program of each clause in $\mathscr{P}$ satisfies $\mathscr{R}$, then $\mathscr{P}$ is **compilable** by the extended algorithm.

It is clear from the definition that the set of compilable programs properly includes all simple programs satisfying $\mathscr{R}$.

Define $\mathscr{A}'$, the **extension** of $\mathscr{A}$, as follows. $\mathscr{A}'$ compiles each clause $B$ in the source program $\mathscr{P}$ into a procedure $\bar{B}$ which, when called, returns the value of its subject variable at $\bar{t}$ in $N^N$. For this, $\mathscr{A}'$ calls on $\mathscr{A}$ to compile the associated simple program $P$ of $B$ as follows:

(1) For a clause $B_1$ with subject $X$ directly nested in $B$, $\mathscr{A}$ compiles the operation $f_X$ into a call of the procedure $\bar{B}_1$. The global variables $Y^{(1)}, \ldots, Y^{(r)}$ of $B_1$, because of the nonstrict nature of $f_X$, have been compiled by $\mathscr{A}$ in a manner which permits their evaluation at $\bar{t}$ on demand. Since $f_X$, and hence the procedure $\bar{B}_1$, has to determine if and when to demand such an evaluation, the code for these evaluations forms the bodies of a set of procedures to be called by $\bar{B}_1$. There is one such procedure for each global variable of $B_1$. This implements the delayed evaluation of the global variables.

(2) For a global variable $Y$ of $B$, the operation $i_Y$ is implemented by a call to the (parameterless) procedure $p_Y$ compiled for the delayed evaluation of $Y$ in the containing clause as defined by (1).

(3) All other constructs are implemented by $\mathscr{A}$ without modifications.

(4) A standard driving program is added in charge of reading input and printing output. The program calls the procedure compiled for the outermost compute clause of the source program requesting the evaluation of the variable **output.**

Intuitively speaking, $\mathscr{A}'$ compiles the outermost clause treating it as a simple program. In this clause, certain variables may be defined by nested clauses, and are each compiled by $\mathscr{A}'$ in the same way. This leads to a recursive implementation of $\mathscr{A}'$.

Recall that the delayed evaluation of the global variables of a clause reflects accurately the nonstrict dependence of the clause on the values of these variables. Also, since the clause derives no special information about the definitions of global variables, they all must be assumed to be defined using unsafe operations. This is due to the manner in which $\mathscr{R}$ has been extended. It should be clear that more complex strategies for extending $\mathscr{R}$ can be formulated which analyze a clause in the context of all containing clauses, distinguishing between safe and unsafe global variables. The principal techniques for defining and implementing the corresponding extension of the basic compilation algorithm would be more intricate but essentially the same.

We now show the correctness of our extension. As before, $|X|_{\bar{t}}$ denotes the value of $X$ at $\bar{t}$ in $N^N$ with respect to the (minimal) solution of the source program.

**Theorem 2.** *Let $B$ be a compute clause with subject $X$ and global variables $Y^{(1)}, \ldots, Y^{(r)}$ which occurs in a compilable program $\mathscr{P}$. Then the procedure $\bar{B}$ compiled by $\mathscr{A}'$ for $B$ returns $|X|_{\bar{t}}$, provided that each procedure $p_i$, $1 \leqq i \leqq r$, when called by $\bar{B}$ for the evaluation of $Y^{(i)}$, returns $|Y^{(i)}|_{\bar{t}}$.*

*Proof.* Observe that the compilability of $\mathscr{P}$ permits us to appeal to the correctness of $\mathscr{A}$ when considering the associated simple program $P$ of $B$. The proof proceeds by induction on the nesting structure of clauses in the program.

**Basis.** $B$ contains no nested clauses: Since the $p_i$ return $|Y^{(i)}|_{\bar{t}}$, the theorem follows from Theorem 1 and the correctness of $\mathscr{A}$.

**Step.** Assume the theorem is true for all compute clauses not containing other clauses nested deeper than $d$, and let $B$ be such that no clause is nested deeper in $B$ than $d+1$: We need to show that all procedures $q_i$ called to evaluate the global variables of directly nested clauses are correct. Then the theorem will follow from the induction hypothesis and the argument of the induction basis.

Consider the associated simple program $P$ of $B$. In it, every nested clause with globals $\bar{Z}$ and subject $V$ appears as the term $V=f_V(\bar{Z})$, where $f_V$ is pointwise. Since the evaluation of the global variables of $B$ is correct by assumption, Lemma 1 and the correctness of $\mathscr{A}$ imply that a correct implementation of the $f_V$ leads to a correct implementation of $P$ and hence of the evaluations of $\bar{Z}$. By Theorem 1 and the induction hypothesis, however, $f_V$ is correctly implemented by its corresponding procedure, hence the procedures $q_i$ evaluating $Z^{(i)}$ must be correct. ■

**Corollary 2.** *Let $\pi$ be the program compiled by $\mathscr{A}'$ for the compilable program $\mathscr{P}$. Then $\pi$ evaluates $|output|_i$ correctly.*

*Proof.* Since the outermost clause has at most input as global variable, the corollary follows from Theorem 2 and the ability of the driving program to evaluate (i.e. read) input correctly, which is trivial. ■

This establishes the correctness of the extension.

## Summary

We have shown how to extend compiling algorithms for simple programs to handle nested compute clauses. Since only very few assumptions about the nature of these algorithms were needed, our method is applicable to a broad spectrum of compiling algorithms. Furthermore, because of the abstraction principle for compute clauses, we could prove correctness of the extension without appealing to technicalities of the proof of the compiler which was extended. All this reduced the complexity of the task.

It is not a far step to make global variables of compute clauses act as formal parameters. In this case we obtain the **mapping clause** of [2]. Because of Theorem 1, the semantic nature of these clauses does not differ in essential ways from the compute clause, and our extension technique can, with very little modifications, be adapted to handle mapping clauses as well. Other clauses of [2], in particular the **transform clause,** appear more difficult to handle.

The syntactic presentation of Lucid, since the definition of basic Lucid in [1], has undergone considerable evolution. In particular, the different clauses of [2] have been recast into a single construct in [12] and [13], whose semantic properties coincide with the compute (mapping) clause if the global variables are **elementary.**

It is perhaps wortwhile to think about the basic approach which this work, as well as any other previous implementation of (parts of) Lucid [5, 6, 14], has taken, and what its implications might be.

Generally speaking, a variable $X$ in Lucid is a complex object which is parametrized by infinite sequences $\bar{t}$ in $N^N$. Any evaluation of a program has to follow an order in which to evaluate parts of $X$ needed for finding the values of output. For programs interpretable in Loop($S$) it is, of course, natural to study the ap-

propriateness of the natural ordering on $N$, i.e., to inquire for which programs it is appropriate always to evaluate $|X|_{t_1}$ before $|X|_{t_2}$, whenever $t_1 < t_2$, provided both values are needed eventually. This has lead to the formulation of syntactic restrictions in [5] and in [6] and to the efficient compilation of such programs. However, this class of programs is restrictive, and the interpreter of [14], which computes the ordering of evaluations during the progression of the program evaluation handles a much larger class of programs.

The derived semantic properties of the compute clause can be shown to imply that, for Comp($S$) interpretations of programs with nested compute clauses, an ordering of $N^N$ is appropriate for the extended class which is, roughly speaking, a reverse lexicographical extension of the ordering of $N$ which $\mathscr{A}$ assumed. The precise formulation of this requires a semantic structure formed with $N^k$, where $k$ is the depth of clause nesting in the source program, rather than using the semantic structures derived by using $N$ and $N^N$, respectively. Research into such orderings could uncover interesting new compilation algorithms, and possibly give hints about augmenting Lucid with control information (in the sense of [11]), which would help a compiler produce a better object program by departing from its usual ordering scheme.

*Acknowledgements.* Ed. Ashcroft and M. O'Donnell made many helpful suggestions on earlier versions. One of the referees suggested an alternate formulation of Lemma 2 and Theorem 1 which led to a clearer presentation and ultimately to conceptually simpler proofs.

## References

1. Ashcroft, E., Wadge, W.: Lucid – A formal system for writing and proving programs SIAM J. Comput. 5, 336–354 (1976)
2. Ashcroft, E., Wadge, W.: Lucid – Scope structures and defined functions. Conference Records: 5th Annl Symp. on Princ. of Progr. Lang., Tucson 1978, 17–23
3. Chirica, L., Martin, D.: An approach to compiler correctness. Proceedings: Int. Conf. on Reliable Software, Los Angeles 1976, Sigplan Notices **10**, 96–103
4. Dahl, O., Dijkstra, E., Hoare, C.: Structured programming London: Academic Press 1972
5. Farah, M.: Correct compilation of a useful subset of lucid. Ph.D. Thesis, Dept. of Comp. Sci., University of Waterloo, 1977
6. Hoffmann, C.: Design and correctness of a compiler for a nonprocedural language. Acta Informat. **9**, 217–241 (1978)
7. Infante, R., Montanari, U.: Proving structured programs correct, level by level. Proceedings: Int. Conf. on Reliable Software, Los Angeles 1976, Sigplan Notices **10**, 427–436
8. Morris, F.: Advice on structuring compilers and proving them correct. Conferene Records: ACM Symp. on Princ. of Progr. Lang., Boston 1973, pp 144–152
9. Vuillemin, J.: Correct and optimal implementation of recursion in a simple programming language. JCSS **9**, 332–354 (1974)
10. Robinson, L., Levitt, K.: Proof techniques for hierarchically structured programs. CACM **20**, 271–283 (1974)
11. Kowalski, R.: Predicate logic as programming language. Information Processing 1974, IFIP Congress Series 569–574
12. Ashcroft, E., Wadge, W.: A logical programming language. RR CS-79-20, Dept. of Comp. Science, Univ. of Waterloo, 1979
13. Ashcroft, E., Wadge, W.: Structured lucid. RR CS-79-21, Dept. of Comp. Science, Univ. of Waterloo, 1979
14. Cargill, T.: Deterministic operational semantics for lucid. TR CS-76-19, Dept. of Comp. Science, Univ. of Waterloo, 1976

Received January 19, 1978 / Revised July 27, 1979