DESIGN AND CORRECTNESS OF A COMPILER FOR LUCID

by
Christoph M. Hoffmann

Research Report CS-76-20
May 1976

Computer Science Department
University of Waterloo
Waterloo, Ontario, Canada

Contents

1. Introduction

Proving the correctness of a compiler is an important and non-trivial problem. Without a correct compiler, no program compiled by it 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 are 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 a year. 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 can be done profitably in parallel.

Proving compiler correctness 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, moreoever, has to be convenient for both proof purposes and implementation strategies. Without such models, a proof cannot be stated, and 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 which has been done was for compilers of languages based on the Lambda Calculus [13, 16] or for small languages isolating a few of the constructs present in imperative languages [7, 14, 17]. Present trends in the work on data structures and programming language design [10, 11] seem to indicate an awareness of the necessity for a formal model of semantics.

Another difficulty in proving compiler correctness, as recognized in [17], 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 which is compiled is Lucid, and was developed by Ashcroft and Wadge [3, 5]. Its motiviation 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. 19], Lucid uses the same denotation for writing and proving properties about programs, thus it is, at the same time, a formal proof system and a programming language. Previously published work has strongly emphasized this point [2, 4].

From a programming point of view it is interesting to note that the flavour of this language is not unlike that of recently

proposed data-flow languages [e.g. 8, 12], in which, as the name suggests, the flow of values through a network of processing nodes is specified and an explicit notion of the 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, which is crucial to proofs of program properties.

A Lucid program may be viewed as a set of assertions about streams of values traversing data paths named by the variables in the program. It is the task of the compiler to analyze the implied data dependencies and deduce from that information a strategy for a coordinated and balanced evaluation of the various streams. Because of existing machine environments, efficiency considerations led to an implementation which does not take advantage of the inherent parallelism in source programs, although this information is made explicit by the analysis performed by the compiler. A few comments on this may be in order.

Along with other similar language proposals, there have been related proposals for new machine architectures specifically designed to exploit the inherent parallelism of programs written in these languages [e.g. 15, 18]. Nevertheless, there are still problems to be solved for accomplishing for a given program a suitable coordination of the various computations and ensuring a

reasonably balanced load across the network. This problem is not likely to be solved automatically by the hardware design and, therefore, has to be solved using information derived from the source program. The analysis performed by the present compiler in an effort to derive a sequential object program can also form the basis for the analysis to be performed for solving the problems of a parallel implementation.

The paper is structured as follows. First, the language and a compilable subset are defined. Since the syntax is rather simple, a BNF notation with additional non context-free constraints was found to define it adequately while, at the same time, possessing great clarity. The semantics is described using the denotational fix-point approach and added comments aid an intuitive understanding of the meaning of the constructs. Fortunately, the semantics of the source language was already completely formalized in [4], otherwise a proof of the compiler could not have been formulated. However, since a fix-point semantics is far removed from efficient implementations, an equivalent operational understanding had to be sought. This is derived in Section 3 of the paper.

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 particularly well-suited because of the simple control structure of the generated code. Because of the nature of the

language Lucid, two easy extensions of the algorithms are possible which enable a compilation of the full subset. These extensions are described and proved correct. Practical considerations in implementing the compiler are addressed and the reasons for the subset restrictions are discussed. These shed further light on the nature of the language constructs, as well as on the design choices made.

2. Compilable Programs and Language Definition

In this section we define the source language Lucid and a subset acceptable to the compiler. Reasons for restricting the compilation to this subset are discussed in Section 7.

The description proceeds in stages: First, the syntax of programs and expressions is given by a cover grammar. The semantics of the constructs is defined following [4] closely. Comments on the intuitive understanding of the definitions complement this exposition. Then, the notion of quiescence is reviewed briefly, and simplifying source transformations are introduced, which make the compilation a more manageable task, and which make explicit certain properties of given programs. Thereafter, syntactically determinable conditions are listed which must be satisfied by programs in the compilable subset.

Sections 3 to 5 develop algorithms for programs without nested blocks, so-called simple programs, and prove the correctness of the algorithms as well as the code compiled. Section 6 generalizes the results extending the algorithms to handle programs with arbitrarily nested blocks, and proving the correctness of the extensions.

Practical considerations are discussed subsequently in detail, as a compiler implemented along the lines of the concepts developed can be quite efficient.

2.1 Expressions

Expressions are formed in the usual manner from variables, constants, parentheses, and the following operators listed below from highest to lowest precedence. Operators of equal precedence bind left-to-right except for the fby operator (pronounced: "followed by"), which binds right-to-left.

1. first, next
2. *, /, rem (remainder division)
3. +, -
4. eq, ne, le, ge, lt, gt
5. not
6. and
7. or
8. if ... then ... else ...
9. asa (pronounced: "as soon as")
10. fby (pronounced: "followed by")

Presently, only types integer and boolean have been implemented, although the implementation of other data types such as string, etc., present no difficulty.

2.1.1 Syntax

The syntax for expressions is given by a cover grammar not reflecting the type restrictions which are listed separately.

```
<expression>        ::= <event exp>
                      | <event exp> fby <expression>
<event exp>         ::= <cond exp>
                      | <event exp> asa <cond exp>
<cond exp>          ::= <disj exp>
                      | if <expression> then <disj exp> else <disj exp>
<disj exp>          ::= <conj exp>
                      | <disj exp> or <conj exp>
<conj exp>          ::= <neg exp>
                      | <conj exp> and <neg exp>
<neg exp>           ::= <rel exp>
                      | not <rel exp>
<rel exp>           ::= <simple exp>
                      | <simple exp><relation op><simple exp>
<relation op>       ::= eq|ne|le|ge|lt|gt
<simple exp>        ::= <term exp>
                      | <simple exp><adding op><term exp>
                      | <adding op><term exp>
<adding op>         ::= +|-
<term exp>          ::= <factor exp>
                      | <term exp><mult op><factor exp>
<mult op>           ::= *|/|rem
<factor exp>        ::= <primary>
                      | first <factor exp>
                      | next <factor exp>
<primary>           ::= <unsigned number>|<name>
                      | <logical value>
                      | (<expression>)
<logical value>     ::= true | false
```

The operators first and next can be applied to expressions of any type, and the resulting expression is of the same type. The operator fby requires that the <event exp> and the <expression> be of the same type; the result type remains unchanged. The asa requires the <cond exp> to be type boolean; the resulting <event exp> has the type of the <event exp> on the left of the asa. All other type restrictions are as in ALGOL 60.

2.1.2 Semantics

Formally, with every variable X is associated as value a mapping from N^N, i.e. the set of infinite sequences of natural numbers, into a domain of values. The set V of values is fixed for our purposes to consist of true, false, $\perp$ (pronounced "undefined"), and all integers. Informally, therefore, a variable may be considered as an infinite sequence of infinite dimensionality over V. Expressions are interpreted as follows.

Definition Let α, β, and γ denote mappings from N^N into V, and $\bar{t} = t_0 t_1 t_2 \ldots \in N^N$. We write $\alpha_{\bar{t}}$ instead of $\alpha(\bar{t})$. Then

(1) $(\omega\ \alpha)_{\bar{t}} = \omega\ \alpha_{\bar{t}}$

where ω is the standard interpretation of not, or <adding op>.

(2) $(\alpha\ \omega\ \beta)_{\bar{t}} = \alpha_{\bar{t}}\ \omega\ \beta_{\bar{t}}$

where ω is the standard interpretation of and, or, or <adding op>, <mult op>, <relation op>.

(3) $(\underline{\text{if}}\ \alpha\ \underline{\text{then}}\ \beta\ \underline{\text{else}}\ \gamma)_{\bar{t}} = \underline{\text{if}}\ \alpha_{\bar{t}}\ \underline{\text{then}}\ \beta_{\bar{t}}\ \underline{\text{else}}\ \gamma_{\bar{t}}$

(4) $(\underline{\text{first}}\ \alpha)_{\bar{t}} = \alpha_{0,t_1t_2\ldots}$

(5) $(\underline{\text{next}}\ \alpha)_{\bar{t}} = \alpha_{t_0+1,t_1t_2\ldots}$

(6) $$(\alpha\ \underline{\text{fby}}\ \beta)_{\bar{t}} = \begin{cases} \alpha_{\bar{t}} & \text{if } t_0 = 0 \\ \beta_{t_0-1,t_1t_2\ldots} & \text{otherwise} \end{cases}$$

(7) $$(\alpha\ \underline{\text{asa}}\ \beta)_{\bar{t}} = \begin{cases} \alpha_{s,t_1t_2\ldots} & \text{if } \beta_{s,t_1t_2\ldots} = \underline{\text{true}},\ \beta_{r,t_1t_2\ldots} = \underline{\text{false}}, \\ & \alpha_{r,t_1t_2\ldots} \neq \perp \text{ for all } r < s; \\ \perp & \text{otherwise} \end{cases}$$

Constants are interpreted as operators of degree 0.

Definition The following operators are special operators:

first, next, fby, asa;

all other operators in the subset are standard operators.

Note that, because of (1) to (3) above, standard operators are 'point-wise', i.e. their result under $\bar{t}$ depends only on their argument values under $\bar{t}$. Thus their compilation can be accomplished by familiar techniques. Special operators, on the other hand, can cause

standard operators to combine argument values of differing index $\bar{t}$. Of these, the index difference effected by first, next, and fby can be determined at compile time.

Also note, that first and next are in part analogous to car and cdr in LISP. The fby provides a two part definition in a manner reminiscent of primitive recursion, while the asa operator is analogous to the μ operator of Recursive Function Theory.

2.2 Programs

2.2.1 Syntax

```
<program>              ::= compute output where <block body>
<block>                ::= <block head> <block body>
<block head>           ::= <compute clause> where
                         | <compute clause> <global references> where
<compute clause>       ::= compute <name>
<global references>    ::= using <name> {, <name>}*
<block body>           ::= <assertion> {; <assertion>}*  end
<assertion>            ::= <sequence definition>
                         | <result definition>
                         | <block>
<sequence definition>  ::= <name> = <expression>
<result definition>    ::= result  = <expression>
```

2.2.2 Semantics

A program is an assertion about the sequence output. Other assertions can be made as part of the program, defining names either as expressions or as blocks, thereby asserting more complex computational relationships. The expression in the result definition, in the context of the block of which it is part, defines the name in the compute clause of the block head.

Any assignment of mappings to the variable names in a program which satisfies the program is a solution of it. The meaning of the program is the minimal solution of it, which was shown to exist in [4].

The global references given in the head of a block B list all names referenced within B and defined in the containing block. The list is constructed by the compiler, rather than given explicitly.

Note, that the sequence in which assertions are made in a block does not affect its meaning.

A sketch of the complete formalism defining the semantics of programs rigorously is added below. For a more detailed exposition the reader is referred to [4].

In order to define exactly what is meant by a solution to a program, the block structure has to be removed by introducing the operators latest and latest^{-1}, which are interpreted as follows.

Let α be a mapping from N^N into V, $\bar{t} = t_0 t_1 t_2 \ldots \in N^N$. Then

(8) $\quad (\underline{\text{latest}}\ \alpha)_{\bar{t}} = \alpha_{t_1 t_2 \ldots}$

(9) $\quad (\underline{\text{latest}}^{-1} \alpha)_{\bar{t}} = \alpha_{0 t_0 t_1 t_2 \ldots}$

Transform the program P as follows: Apply $\underline{\text{latest}}^{-1}$ to each expression defining result, and substitute the name in the compute clause of the corresponding block head for result. Remove the block head and corresponding end after prefixing each reference global to the block within the block and every nested block by latest.

As was shown in [4], the new program is equivalent to the old one, and furthermore, each variable X is now defined by

X = <expression>

It is now possible to understand a program P as a function F_P on the mappings provided by the variables, so that the meaning of a program is the minimal fixpoint of the equation

$$\bar{X} = F_P(\bar{X})$$

where $\bar{X}$ denotes the vector of all variable names in P.

For an example of a program in latest notation see Example 2.1. Preference has been given to the block structured notation because firstly it was found to greatly improve the readability of programs, and secondly, it does not permit writing certain pathological programs. A thorough discussion of these points, however, is beyond the scope of this paper.

2.3 Source Simplifications

Several source to source transformations are described which are applied by the compiler in order to reduce the complexity of the source program.

2.3.1 Expression Simplification

Expressions are simplified to be in one of three forms, basically by introducing new variable names.

Definition An <expression> is a formula, if first and next are the only special operators occurring in it.

By defining new names, all expressions are to be in the following form:

(1) <formula>

(2) <formula> fby <formula>

(3) <name> asa <name>

where each name in (3) must, in addition, be defined by a formula and referenced only once, namely by the asa expression.

Example 2.1 Consider the following Lucid program:

```
compute output where
   c      = 0 fby c+1;
   sum    = 0 fby sum+root;
   m      = first input;
   n      = next  input;
 result = sum asa c eq m;
 compute root where
    cc     = 0 fby cc+1;
    y      = 1 fby y+2*cc+3;
  result  = cc asa y gt n
 end
 end
```

After performing the simplifications described, the new program is

```
compute output where
    c      = 0 fby c+1;
    sum    = 0 fby sum+root;
    m      = first input;
    n      = next  input;
   result = t1 asa p1;
    t1     = sum;
    p1     = c eq m;
    compute root where
        cc     = 0 fby cc+1;
        y      = 1 fby y+2*cc+3;
      result = t2 asa p2;
        t2     = cc;
        p2     = y gt n
    end
end
```

The equivalent version of the first program using latest notation is:

```
     c = 0 fby c+1;
   sum = 0 fby sum+root;
     m = first input;
     n = next  input;
output = latest^-1 (sum asa c eq m);
     y = 1 fby y+2*cc+3;
    cc = 0 fby cc+1;
  root = latest^-1(cc asa y gt latest n)
```

2.3.2 Constructing the Global References

By analyzing the reference structure of the source program, the compiler constructs for each block with aid of a symbol table a <global references> clause whenever the list of names in it is not empty. Given a block B_1, the global references list in its block head will contain precisely those names which are defined in the immediate ancestor block, and which are referenced in B_1 or any block nested in B_1.

Example 2.2 For the program of the previous example a clause

"using n"

will be inserted between "compute root" and "where".

In the following programs are assumed to have been simplified in the manner described above.

2.4 Quiescence

Within a block B an expression may satisfy the equation

first E = E.

If this can be deduced syntactically, E is said to be quiescent.

Definition An expression E is quiescent in a block B if

(a) E is a constant, or a name defined in a block global to B; or

(b) E is a name defined by an expression which is quiescent in B; or

(c) E is defined by the compute clause of a block B_1 immediately contained in B, and the global references list of B_1 is empty or contains only names defined by quiescent expressions; or

(d) $E = (\omega, F_1, .., F_r)$, where ω is a standard operator of degree r and the expressions $F_1, .., F_r$ are all quiescent in B; or

(e) E = first F for some expression F; or

(f) $E = F_1$ asa F_2 for some expressions F_1 and F_2; or

(g) E = next F for some expression F which is quiescent in B.

Nothing else is quiescent in B.

Example 2.3 Consider the first two programs of Example 2.1. In their outer block

first input, sum asa c eq n, m

are all quiescent. Also, in their inner blocks, n is quiescent. Note, however, that n is not quiescent in the outer block.

That the expressions which are defined above satisfy first E = E is immediate except for part (c). See Corollary 6.2 for this.

2.5 Syntactic Requirements

A <program> is in the compilable subset, if all of the following are satisfied:

2.5.1 All <names> referenced must be defined exactly once, either by a <sequence definition>, or by a <compute clause> in a block head, except the name input which is considered to be implicitly defined in the outermost block.

2.5.2 A name defined in an inner block may not be referenced in an outer block. Names obey the ALGOL 60 scope rules. A name in a compute clause of a block B_1 is defined in the block B immediately containing B_1.

2.5.3 Each block must contain exactly one result definition. The expression in the result definition must be quiescent.

2.5.4 Each name is either of type integer or boolean, and its definition and use must be consistent with its type.

2.5.5 Construct for each block a directed graph labelling its edges forming a closed cycle must have a negative sum.

2.6 Construction of the Dependency Graphs

For each block of the source program introduce new names simplifying all expressions to be in one of the following five forms:

(i) first <name>

(ii) next <name>

(iii) <name> asa <name>

(iv) <name> fby <name>

(v) <pointwise expression>

where a pointwise expression is an <expression> in which no special operators occur.

The nodes of the graph for a block B are all names defined in B. If X is defined by

(a) first A,

draw an arc from X to A labelled 0;

(b) next A,

draw an arc from X to A labelled 1;

(c) A asa B,

draw arcs from X to A and from X to B, both labelled ∞;

(d) A fby B,

draw an arc from X to B labelled -1, and from X to A labelled 0;

(e) X = <pointwise expression>,

draw an arc from X to each operand A in the expression labelled 0;

(f) compute X using <list> where ..,

draw an arc from X to each A in the <list> labelled 0.

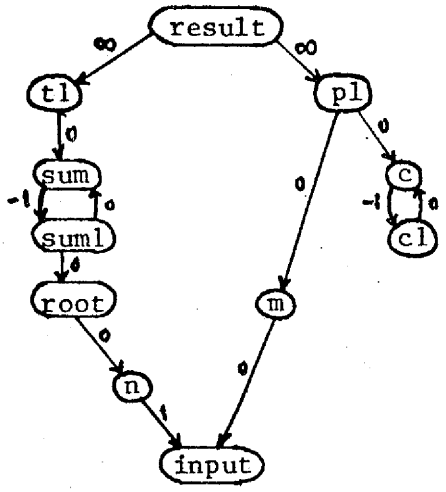
If a name is not defined within that block, no arc is drawn.

Example 2.4

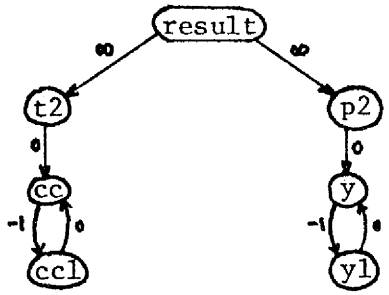
The additional simplifications described transform the program of Example 2.3 into:

```
compute output where
  c      = 0  fby c1;
  sum    = 0  fby s1;
  m      = first input;
  c1     = c+1;
  s1     = sum + root;
  n      = next input;
  result = t1 asa p1;
  t1     = sum;
  p1     = c eq m;
  compute root using n where
    cc     = 0 fby cc1;
    y      = 1 fby y1;
    cc1    = cc+1;
    y1     = y+2*cc+3;
    result = t2 asa p2;
    t2     = cc;
    p2     = y gt n;
  end
end
```

The graphs for the two blocks are:



(outer block)



(inner block)

Note that no arc is drawn from y to n in the graph above, since n is not defined in the same block. Observe also, that all cycles have an edge sum of -1.

3. Modified Programs and Defining Terms

Recall that the meaning of a Lucid program is defined by a fix-point semantics. Since, for computation 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 fix-point interpretation and an equivalent operational understanding of the constructs. To this end two mappings, ϕ_n and τ_n, where $n \geq 0$, are introduced. At the same time, ϕ_n and τ_n effect a removal of the operators first and next, and are instrumental in specifying a program transformation needed to arrive at an operational interpretation of programs.

For the sake of a clear presentation, programs are assumed to contain no nested blocks. It is then justified to interpret each variable as a mapping from N into V, i.e. as an infinite sequence over V. In Section 6 the formalism is extended to cover the general case.

Two operators, :i and <i>, where $i \geq 0$, 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. α maps 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. In fact, with an extended definition, $X{:}i = \underset{\sim}{\text{first}}\ \underset{\sim}{\text{next}}^{i}\ X$, and $X\langle i\rangle = \underset{\sim}{\text{next}}^{i}\ X$.

Definition An expression formed from constants, parentheses, standard operators and variable names to each of which one suffix operator :i or <i> is applied is called a term. Expressions formed in the above manner, but also admitting the operators fby and asa, are called extended terms.

Example 3.1

5, X:0*(Y<1>+Y<0>), X<3>

are all terms. The following is an extended term

X<0> asa (Y:0 fby X<2>) eq 45

whereas

X<3>:4, X+Y:3

are neither terms nor extended terms.

The mappings ϕ_n and τ_n, where $n \geq 0$, which map 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

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

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

(c) If F is an expression, then

$$\phi_n(\underset{\sim}{\text{first}}\ F) = \phi_0(F)$$

$$\tau_n(\underset{\sim}{\text{first}}\ F) = \phi_0(F)$$

$$\phi_n(\underset{\sim}{\text{next}}\ F) = \phi_{n+1}(F)$$

$$\tau_n(\underset{\sim}{\text{next}}\ F) = \tau_{n+1}(F)$$

(d) If F and G are expressions, then

$$\phi_0(F\ \underset{\sim}{\text{fby}}\ G) = \phi_0(F)$$

$$\tau_0(F\ \underset{\sim}{\text{fby}}\ G) = \phi_0(F)\ \underset{\sim}{\text{fby}}\ \tau_0(G)$$

$$\phi_{n+1}(F\ \underset{\sim}{\text{fby}}\ G) = \phi_n(G)$$

$$\tau_{n+1}(F\ \underset{\sim}{\text{fby}}\ G) = \tau_n(G)$$

(e) If F and G are expressions, then

$$\phi_n(F\ \underset{\sim}{\text{asa}}\ G) = \tau_0(F)\ \underset{\sim}{\text{asa}}\ \tau_0(G)$$

$$\tau_n(F\ \underset{\sim}{\text{asa}}\ G) = \tau_0(F)\ \underset{\sim}{\text{asa}}\ \tau_0(G)$$

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

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

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

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

$$(|\phi_n(E)|)_t = (|E|)_n$$

$$(|\tau_n(E)|)_t = (|E|)_{t+n}$$

Proof (By induction on the structure of E)

(a) If E is a variable name, the theorem is trivial.

(b) Let ω be an r-ary standard operator, $F^{(1)}, .., F^{(r)}$ expressions, and assume $E = (\omega, F^{(1)}, .., F^{(r)})$. Then

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

The argument for τ_n is analogous.

(c) Let E = first F, where F is an expression, then

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

and analogously for τ_n. Furthermore, with E = next F,

$$\begin{aligned}(|\phi_n(E)|)_t &= (|\phi_{n+1}(F)|)_t \\ &= (|F|)_{t+n+1} \qquad \text{(ind. hyp.)} \\ &= (|E|)_{t+n} \qquad \text{(def. of next)}\end{aligned}$$

and analogously for τ_n.

(d) Let E = F fby G, F and G expressions. Then

$$(|\phi_0(E)|)_t = (|\phi_0(F)|)_t$$
$$= (|F|)_0$$
$$= (|E|)_0 \qquad \text{(def. of fby)}$$

$$(|\tau_0(E)|)_t = (|\phi_0(F) \text{ fby } \tau_0(G)|)_t$$

Two cases arise,

(1) t=0, then

$$(|\phi_0(F) \text{ fby } \tau_0(G)|)_t = (|\phi_0(F)|)_t \qquad \text{(def. of fby)}$$
$$= (|F|)_0$$
$$= (|E|)_{t+0} \qquad \text{(def. of fby)}$$

(2) t>0, then

$$(|\phi_0(F) \text{ fby } \tau_0(G)|)_t = (|\tau_0(G)|)_{t-1} \qquad \text{(def. of fby)}$$
$$= (|G|)_{t-1}$$
$$= (|E|)_t \qquad \text{(def. of fby)}$$

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

(e) Let E = F asa G, finally, F and G expressions. Then, by definition of the asa,

$$(|E|)_t = \begin{cases} (|F|)_s & \text{if } (|G|)_s = \underline{\text{true}},\ (|G|)_r = \underline{\text{false}} \text{ and} \\ & \quad (|F|)_r \neq \perp \text{ for all } r<s; \\ \perp & \text{otherwise.} \end{cases}$$

Since, by induction hypothesis, $(|F|)_t = (|\tau_0(F)|)_t$ and $(|G|)_t = (|\tau_0(G)|)_t$, clearly

$$(|E|)_t = (|\tau_0(F) \text{ asa } \tau_0(G)|)_t,$$

and, by definition of asa,

$$(|E|)_t = (|E|)_0 \quad \text{for all } t,$$

from which the theorem follows.

□

Intuitively, ϕ_n and τ_n have the same effect on expressions as :n and <n> have on variables. Yet they are not simple extensions of theses operators, as they, at the same time, accomplish a mapping into extended terms which removes all operators first and next.

The notion of transformed program is introduced. The transformed program of a given (simplified) program P is an infinite equivalent program containing only extended terms.

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

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

$$X{:}i = \phi_i(E);$$

$$X\langle i\rangle = \tau_i(E);$$

for all $i \geq 0$.

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

$$X{:}0 = \phi_0(E);$$

$$X\langle 0\rangle = \phi_0(E) \text{ fby } X\langle 1\rangle;$$

$$X{:}i = \phi_{i-1}(F);$$

$$X\langle i\rangle = \tau_{i-1}(F);$$

for all $i > 0$.

(iii) P' contains no other statements.

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, and as it has to be quiescent by syntax requirements, usually a subprogram of P' is sufficient to define it.

Definition Let P' be the transformed program of P. 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 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 remainder of this section is devoted to deriving the result that P_0 is finite for programs P in the subset. Intuitively speaking, the subset restrictions have been designed primarily to imply this result, since the finiteness of P_0 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[i] as new variable names neglecting that :i and <i> have been interpreted as operators. When doing so, in order to avoid confusion, we shall denote these new variables by X.i and X[i], respectively.

Definition Given a variable name X, X.i and X[i] are qualified names derived from X ($i \geq 0$). Furthermore, X.i and X[i] name the terms X:i and X<i>, respectively.

Definition Let X be a variable name occurring in a program P, E and F formulae. The defining term of a qualified name derived from X is

$$\text{def}(X.i) = \begin{cases} \bar{\phi}_0(E) & \text{if } X = E \text{ fby } F,\ i=0 \\ \bar{\phi}_{i-1}(F) & \text{if } X = E \text{ fby } F,\ i>0 \\ \bar{\phi}_i(F) & \text{if } X = F \\ \bar{\tau}_0(E) \text{ asa } \bar{\tau}_0(F) & \text{if } X = E \text{ asa } F \end{cases}$$

$$\text{def}(X[i]) = \begin{cases} \bar{\phi}_0(E) & \text{if } X = E \text{ fby } F,\ i=0 \\ \bar{\tau}_{i-1}(F) & \text{if } X = E \text{ fby } F,\ i>0 \\ \bar{\tau}_i(F) & \text{if } X = F \\ \bar{\tau}_0(E) \text{ asa } \bar{\tau}_0(F) & \text{if } X = E \text{ asa } F \end{cases}$$

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

Observe the close correspondence between defining terms and the statement transformations defined earlier. The only deviation, the defining term of X[0] where X is defined by a fby expression, is for a technical reason which will become clear in Section 4.

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

The following theorem relates the edge labelling conventions of Section 2.6 to the dependencies among qualified names.

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

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

Basis k=1, α depends directly on β.

(a) Y is the operand of a pointwise expression defining X. Therefore s=0, and, by definition of $\bar{\phi}_n$ and $\bar{\tau}_n$, i=j.

(b) X = first Y, thus s=0. Obviously j=0, and since $i \geq 0$, $j-i \leq 0$.

(c) X = next Y. Therefore, s=1, and j=i+1.

(d) X = Y fby Z. So, s=0, i=0, j=0.

(e) X = Z fby Y. So, s=-1, i>0, and j=i-1.

(f) X = Y asa Z or X = Z asa Y. Since $s=\infty$, the theorem is trivial.

Step $k \rightarrow k+1$:

Assume the theorem is true for a path of length k, and that $\alpha \bullet\!> \beta$ with a path of length k+1. There must be some Z other than X and Y on the path from X to Y, so there will be some γ = Z[r] or Z.r such that $\alpha \bullet\!> \gamma$ and $\gamma \bullet\!> \beta$. By hypothesis,

$$r-i \leq s_1$$

$$j-r \leq s_2$$

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 Suppose P_0 is infinite. Then there will be qualified names α and β such that

α = X[i] or X.i

β = X[j] or X.j

with $\alpha \cdot\!\!> \beta$ and j>i, since there are only finitely many variable names. in P. By the previous Theorem 3.3 there is a path from X to X in the dependency graph, i.e. a cycle, with an edge label sum $s \geq j-i > 0$, contrary to the subset syntax.

□

4. Compilation of Simple Goals

Given a program, subsets of defined variables can be isolated which are self-contained with respect to their reference structure, that is, each variable definition in the subset is made in terms of other variables also defined in the subset. Such subsets are called nests.

Given a variable R = E asa F, then R can be evaluated 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 the value of $|R|_t$ for all t (note that R is quiescent).

Thus an iteration evaluating expression is done to find the value of R. For the evaluation of E and F other variables may have to be evaluated, more accurately, exactly those variables forming the smallest nest containing R.

Because of the labelling conventions of the dependency graph, 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 this section is concerned with compiling them. First the algorithms are developed and basic properties of them are shown. A running example illustrates the compilation. As target language a bastard ALGOL is used for readability, -- compiling machine level code instead is easily

accomplished by well-known techniques. Thereafter, the correctness of the generated code is proved.

The algorithms of this section must be considered basic, since only small modifications extend them to compile the full subset.

The code to be compiled is structured to consist of a prelude followed by a loop. The prelude is essentially a straight-line program containing computations which need to be done just once, e.g. computations of names of the form X.i. The loop iterates the evaluation of non-quiescent expressions.

Because fby expressions are evaluated by computing a special case once and thereafter iterating a general case, and because of the interaction of different fby expressions, the loop may have to be unrolled a number of times accommodating all special case computations in the enlarged prelude, until the different fby expressions 'catch up' and are all computed from their general case. This is accomplished as follows.

Given a simple goal R = E asa F, Algorithm A4.1 constructs an operand list containing all qualified names which need to be computed in order to evaluate E and F. If any name occuring in the list has to be computed initially from the first part of a fby expression, then the loop is unrolled once, and a new list is constructed. This process is repeated resulting in the operand lists $L_0, L_1, \ldots, L_n$ until all names defined by a fby expression are computed in the last list from the general case. At the same time the existing dependencies

among all names are recorded. The intuitive understanding of the operand lists is that L_i consists of all those qualified names which need to be computed for the i-th and subsequent iterations, but not for the iterations 0,1,...,i-1.

Algorithm A4.2 linearizes the dependency graph constructed by A4.1, thereby determining how the various names can be computed sequentially. Because of the syntax requirements this can always be accomplished. Algorithm A4.3 constructs a schedule for code generation from the output of A4.2, and Algorithm A4.4 generates the actual (unoptimized) code.

Note that, from a formal point of view, a part of the transformed program P' is compiled after replacing each term X:i and X<i> by the corresponding qualified name. However, P' is never constructed explicitly, rather it is derived incrementally from the original program.

Definition A set of variables $X^{(1)},\ldots,X^{(r)}$ is a nest, if the definition of each of the $X^{(i)}$ 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 if the definitions of the variables in the smallest nest containing X do not reference any goal.

Example 4.1 Consider the following segment of a simple program:

```
R = E asa F;
E = X;
F = Y gt 25;
X = 0 fby X+1;
Y = 1 fby Y+Z;
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 with Example 4.5 listing the compiled code.

Algorithm A4.1

Input: A simple goal R = E asa F

Output: Operand lists $L_0,\ldots,L_n$ containing the qualified names which are needed to evaluate R correctly. Also, for each item in L_i, $0 \leq i \leq n$, a dependency list is constructed.

1. [Initialize]

   Set n to zero.

2. [Initialize next operand list]

   Set L_n to be the list <E[n], F[n]>.

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 β 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 such that 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 as input, 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 is defined by a fby expression.

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

Proof Observe first, that a new list L_{r+1} is constructed only if L_r contains a name X[0] where X is defined by a fby expression. Since a new name is appended to a list L_r only if it does not occur in $L_0,\ldots,L_r$ already, the n of A4.1 is bounded by 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 in the smallest nest containing R, there will be qualified names α and β such that

$\alpha \bullet\!> \beta$,

α = X[i] or X.i,

β = X[j] or X.j, and $j > i$.

By Theorem 3.3, there is a cycle in the dependency graph involving the node X with edge label sum

$s \geq j-i > 0$

contrary to subset syntax. Hence each list L_r is finite.

□

Algorithm A4.2 is given below. It is essentially a topological sort with added constraints.

Algorithm A4.2

Input: Operand lists $L_0,\ldots,L_n$ constructed by A4.1 for a simple goal R in a subset program

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 items select one by applying the following criteria in sequence.

   As soon as one criterion is applicable go to Step 5.

   4.1: If some X.j is admissible, select the first such X.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 to S_m .

6. [Sort completion condition]

If L_m is not empty, goto Step 3;

otherwise increment m by 1 and go to Step 2.

Example 4.3 The sorted lists for input L_0, L_1, L_2 of the previous example will be:

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]>

Before proving that A4.2 always halts the notion of close dependency is introduced.

Definition A qualified name α closely depends on another qualified name β (in symbols α *> β), if

(i) α •> β, and

(ii) in the direct dependency chain

$\alpha = \alpha_1$ •> α_2 •> ... •> $\alpha_k = \beta$

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 direct dependency chain)

Basis X[i] directly depends on Y[j]. If X is defined by a fby expression, then, since def(X[0]) depends only on names of the form Z.m, i must be greater than 0. Consequently,

$$def(X[i]) = \bar{\tau}_r(E)$$

$$def(X[i+1]) = \bar{\tau}_{r+1}(E) \quad \text{for some formula E,}$$

from which the lemma follows.

Step Split the dependency chain such that

X[i] *> Z[r] *> Y[j].

By induction hypothesis

X[i+1] *>Z[r+1]

Z[r+1] *>Y[j+1]

hence X[i+1] *>Y[j+1].

□

Theorem 4.3 Algorithm A4.2 always halts.

Proof Observe that none of the lists L_i is empty, and that no item in L_i can depend on another item in L_j where $j > i$. Thus it suffices to show that the sort succeeds for any list L_r after the lists $L_0,\ldots,L_{r-1}$ have been sorted. Suppose, therefore, that L_r cannot be sorted. Two cases must be considered:

1) No items were found admissible.

2) No admissible item qualified for selection in Step 4.

Case 1: There must exist a circular dependency, i.e. there is some α in L_r such that $\alpha \cdot\!\!> \alpha$. By Theorem 3.3 this contradicts the subset syntax requirements.

Case 2: Observe first that any item of the form X.j depends only on items of the same form. Hence, if no item can be selected, then only names of the form X[j] are left in L_r and all admissible items violate Criterion 4.2.

Let $\alpha_1,\ldots,\alpha_m$ be all admissible items and assume that none can be selected. Without loss of generality, assume that $\alpha_{p+1},\ldots,\alpha_m$ cannot be selected because of some items in $\alpha_1,\ldots,\alpha_p$. 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

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

Furthermore, since α_{p+1}, .., α_m cannot be selected because of

$\alpha_1 \ldots \alpha_p$,

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

where k is such that $X^{(k)} = X^{(j)}$, $t_j > t_k$, and $k \leq p$.

Since the $\beta_1 \ldots \beta_p$ are inadmissible, they depend on some of the $\alpha_1 \ldots \alpha_m$. Let f(j) be the smallest k such that

$$\beta_j \cdot\!> \alpha_k \qquad 1 \leq j \leq p,\ 1 \leq k \leq m,$$

and oserve that β_j closely depends on α_k. Define g by

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

Because of the ordering of the admissible items, g is well-defined. Consider the sequence 1, g(1), $g^2(1)$, .., $g^p(1)$. Since $g(t) \leq p$ by definition, at least one value $j \leq p$ occurs twice, so that there is a $k \leq p$ such that $g^k(j) = j$. By assumption,

$$\begin{aligned}
X^{(j)}[i_j] &\ *\!> X^{(f(j))}[i_{f(j)} + t_{f(j)}] &&= X^{(g(j))}[i_{g(j)} + \bar{t}_1] \\
X^{(g(j))}[i_{g(j)}] &\ *\!> X^{(f(g(j)))}[i_{f(g(j))} + t_{f(g(j))}] &&= X^{(g^2(j))}[i_{g^2(j)} + \bar{t}_2] \\
&\ \ \vdots \\
X^{(g^{k-1}(j))}[i_{g^{k-1}(j)}] &\ *\!> X^{(g^k(j))}[i_{g^k(j)} + \bar{t}_k] &&= X^{(j)}[i_j + \bar{t}_k]
\end{aligned}$$

By repeated application of Lemma 4.2, therefore,

$$X^{(j)}[i_j] \ *\!> X^{(j)}[i_j + \bar{t}_1 + .. + \bar{t}_k].$$

In conjunction with Theorem 3.3 this implies a violation of the subset syntax contrary to assumption. Therefore, at least one item can be selected from the $\alpha_1 \ldots \alpha_m$.

□

The following algorithm constructs a schedule for code generation from the sorted lists. This involves combining the lists and marking the places into which tests for loop termination have to be put, and determining the exact loop limits. Algorithm A4.4 generates target code from this output.

Algorithm A4.3 (Construct Schedule S)

Input: Sorted Lists L_0, .., L_n from A4.2.

Output: Schedule S for Code Generation.

1. [Construct first part of S]

   Let T_i denote the list containing the special symbol $\#_i$ only.

   Set S to be the concatenation of L_0, T_0, .., L_{n-1}, T_{n-1}.

2. [Determine loop]

   Let α_1, .., α_m be all items in L_n.

   Scan L_n marking an item X[i] as 'in-loop' if X[i+1] is not in L_n. Thereafter, set p to be the smallest j such that α_j is labelled 'in-loop'.

   Split L_n into the lists

   $L' = \langle \alpha_1, .., \alpha_p \rangle$ (empty if p=1)

   $L'' = \langle \alpha_{p+1}, .., \alpha_m \rangle$

3. [Complete schedule]

   Append to S the lists L', T', L", T_n in sequence, where T' contains the symbol % only.

   Thereafter stop.

Example 4.4 The schedule constructed from the sorted lists of the previous example is

S = <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], $\#_2$>.

Algorithm A4.4 (Code Generation)

Input: Schedule S = <α_1, .., α_r> from A4.3.

Output: Code evaluating the simple goal R = E asa F which was input to A4.1.

1. [Initialize iteration count and code S]

   Emit "begin t ← 0; "

   Scan S and code each item α_i as prescribed by Step 2.

   Thereafter go to Step 3.

2. [Code item α in S]

   If α is a qualified name, then

   emit " α ← β; ", where β is def(α).

   If α is $\#_j$, then

   emit " if F[j] then begin R.0 ← E[j];
   goto L;
   end; "

   If α is %, then

   emit " repeat forever begin ".

3. [Code iteration count increment and window shifts]

   Emit " t ← t+1; "

   Scan all qualified names α following % in S and perform the following:

If a name X[i] occurs in def(α) which does not follow the % in S, then generate instructions

```
X[i]   ← X[i+1];
X[i+1] ← X[i+2];
   ⋮
X[j-1] ← X[j];
```

where j is the smallest k > i such that X[k] is in S following the %.

Collect all instructions so generated, delete all 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 shows that to each X[i] found in Step 3 of the algorithm, there is some X[k] with k>i in S following the %.

Example 4.5 The code generated by A4.4 from the schedule of the previous example is:

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

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

The correctness proof of the object program rests on certain properties of the lists constructed throughout the compilation. These properties are derived first and summarized by Theorem 4.7. In particular, the theorem proves that no value is computed, unless the next iteration is necessary, and the computation required by it. Thus, there are no 'look-ahead' computations. Algorithms which do a certain amount of look-ahead computation can be devised and compile simpler code, but they would restrict the domain of source programs which can be compiled correctly.

Furthermore, if some X[i] is referenced by a defining term within the loop, but is not computed within the loop previously, Theorem 4.7 proves the existence of names X[i+1], .., X[j] such that X[j] is computed within the loop, and the X[i+1], .., X[j-1] have been initialized before computing X[j] for the first time. As a consequence, the new values for X[i] can be obtained subsequently by a 'window shift', that is, by

```
X[i]   ← X[i+1];
        :
X[j-1] ← X[j];
```

All these names may be conceptualized to be a fixed-size queue (window) buffering j-i+1 consecutive component values of the sequence X until no longer needed.

The following results derive some properties of the dependency relation among qualified names, and their effect on the list construction.

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

Proof If X[i] is in L_r, then E[r] or F[r] closely depend on it. By Lemma 4.2, E[r+1] or F[r+1] closely depend on X[i+1], hence X[i+1] must occur in L_0, .., L_{r+1} .

□

Corollary 4.5 Let L_0, .., L_n be the operand 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, .., L_{n-1}, $i_0 > j$.

Proof Obvious.

Next, the structure of the sorted lists output by A4.2 is derived.

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 of the form X.i in S_r precedes every name of the form 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 Observe that any name of the form X.i depends only on names of the same form. Therefore, (1) follows from Criterion 4.1 of A4.2. Furthermore, (2) follows from Criterion 4.2. For (3) observe that, because of the construction of the corresponding list L_r by A4.1, for any α in S_r other than E[r] and F[r], E[r] or F[r] depend on α.

□

Theorem 4.7 Let S be the schedule constructed by Algorithm A4.3 from the sorted lists $S_0,\ldots,S_n$ output by A4.2. Let $L_0,\ldots,L_n$ be the corresponding operand lists constructed by Algorithm A4.1, and denote by S" that part of S which follows the special symbol %. Then the following is true:

(1) If α is a qualified name in S and α •> β, then β is in S and precedes α.

(2) If β is a qualified name in S and is different from E[r] and F[r] for all $r \leq n$, then there is a qualified name α such that α •> β. Furthermore, if β is in S_r, then so is α.

(3) If α is a qualified name in S" and X[i] occurs in def(α) but is not in S", then there are names

X[i+1],...,X[j]

in S such that X[j] is in S" and the X[i+1],...,X[j-1] all precede X[j].

Proof (1) Let $\alpha = \alpha_1$ •> α_2 •> ... •> $\alpha_k = \beta$ be the direct dependency chain from α to β. Since α is in S, there is some S_r and consequently L_r such that α is in L_r. By the definition of direct dependency, α_{i+1} occurs in def(α_i) for all $i < k$, thus, by construction of L_r, α_{i+1} is in $L_0,\ldots,L_r$, for all $i < k$. β precedes α because Algorithm A4.2 is a topological sort.

(2) The existence of α is immediate. That α must be in S_r follows from the construction of L_r by A4.1.

(3) If X[i] is not in S_n, then, by repeated application of Lemma 4.4, there are names

X[i+1],...,X[k]

in $S_0,\ldots,S_n$ such that X[k] is in S_n, but the X[i+1],...,X[k-1] are not, and therefore precede X[k] in S.

Furthermore, there are names

X[k], X[k+1],...,X[j]

in S_n (possibly k=j), such that X[j+1] is not in S_n. Then X[j] is labelled 'in-loop' by Algorithm A4.3 and consequently is in S". By Lemma 4.6 the X[k],...,X[j-1] all precede X[j] in S_n and therefore in S.

If 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 E[0], F[0],...,E[n], F[n] depend as well as E[i] and F[i]. Furthermore, for every $r \leq n$, $S_0,\ldots,S_r$ contain exactly E[0], F[0],...,E[r], F[r] and the names on which they depend. The sequence in which the names are arranged in S is compatible with the existing dependencies.

The third part of the theorem is instrumental in proving properties of the instructions generated by Step 3 of Algorithm A4.4.

At this point it is convenient to review the structure of the object language. The following constructs are available:

$\alpha \leftarrow E$	α is assigned the value of E. α is a variable which can store one element of V, the value domain, and E is an expression over such variables, constants and standard operators in the subset. Note, that the value of E cannot be $\perp$ unless at least one variable in it has that value.
begin ... end	delimiters for compound statements and blocks.
repeat forever S	an infinite loop with S as its body. Control may exit from the loop by a goto instruction.
goto L	unconditional transfer of control to the label L.
if α then S	if α has the value true, then S is executed, otherwise do nothing.
procedure P S	non-recursive procedure P with body S. A call of P is equivalent to the substitution of S at the point of call.

The semantic concepts of the object language are quite familiar and easily axiomatized. We assume that a correct interpreter is available for it, which is justified by other research [e.g. 7, 17].

Recall that the schedule S is the concatenation of the lists

$$S_0, T_0, \ldots, S_{n-1}, T_{n-1}, S', \langle\%\rangle, S'', T_n$$

where $T_i = \langle \#_i \rangle$ and $S_n = S'\ S''$. Consequently, the generated code has 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 coded 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.

Let ρ be a solution of the source program, and denote by $|X|_\rho$, $|X{:}i|_\rho$, and $|X\langle i\rangle|_\rho$ the values of X, X:i and X<i> for the solution ρ, respectively. Let $(| \ldots |_\rho)_t$ denote the t component of the value $| \ldots |_\rho$, and recall that, for simple programs, $t \in N$ rather than $t \in N^N$.

σ denotes the minimal fix-point solution of the source program.

Let h be the homomorphism which maps expressions over qualified names into terms by replacing each qualified name by its corresponding term. E.g. h(X.i) = X:i, and h(X[i]) = X<i>.

For proving the correctness of the object program we prove first 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 program.

Theorem 4.8 Let r be the value of t before executing the j-th assignment statement to 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: Assume that the j-th assignment to α is executed for the first time. We show the theorem by induction on j.

Basis: j = 1.

Clearly r=0 and the assignment to α is of the form

$$\alpha \leftarrow def(\alpha).$$

Because of the topological sort, no qualified name is referenced by $def(\alpha)$, and its value is therefore constant. By Theorem 3.1, for any solution ρ, since r=0,

$$(|h(\alpha)|_\rho)_r = (|h(def(\alpha))|_\rho)_r$$

and thus, in particular, for the minimal solution σ

$$(|h(\alpha)|_\sigma)_r = (|h(def(\alpha))|_\sigma)_r$$

consequently

$$v(\alpha) = v(def(\alpha)) = (|h(def(\alpha))|_\sigma)_0 = (|h(\alpha)|_\sigma)_r$$

Step The theorem is true for the first j-1 assignments to qualified names. The j-th assignment can be of one of three forms:

$\alpha \leftarrow def(\alpha)$ (case a)

R.0 ← E[k] (case b)

X[i]← X[i+1] (case c)

Case a

By hypothesis and Theorem 4.7, for each qualified name β in $def(\alpha)$

$(|h(\beta)|_\sigma)_r = v(\beta)$

(Neither t nor β can have been assigned since the first time β was assigned by the k-th assignment, k< j).

A reasoning analogous to the argument above shows that

$v(def(\alpha)) = (|h(def(\alpha)|_\sigma)_0 = (|h(\alpha)|_\sigma)_r = v(\alpha)$.

Case b

The assignment is to R.0, and therefore, by the construction of the code, F[k] = **true**. Since F[0], .., F[k-1] have been tested before F[k], and because of the goto L instructions, the values of the F[0], .., F[k-1] are all false. By hypothesis therefore, for r=0 (i.e. F[k] is tested for the first time),

$(|F<j>|_\sigma)_0$ = false for all j < k,

$(|F<k>|_\sigma)_0$ = true

$(|E<k>|_\sigma)_0 = v(R.0)$

since $(|X<s>|_\rho)_t = (|X|_\rho)_{t+s}$,

$(|R:0|_\sigma)_0 = v(R.0)$

by definition of asa.

For r >0, i.e. k=n and F[n] is tested for the r+1 st time, see Case b in Part 2 of the proof.

Case c

The assignment is of the form X[i] ← X[i+1] and occurs in Q. Therefore, r=1, and, by the construction of Q, X[i+1] cannot have been assigned in Q preceding this assignment. By Theorem 4.7, X[i+1] has been assigned once by a statement preceding the t ← t+1, so that, by hypothesis,

$$v(X[i+1]) = (|X<i+1>|_\sigma)_0 = (|X<i>|_\sigma)_1$$

from which case c follows.

Part 2 Assume that the j-th assignment to α is executed for the n-th time where n>1. Because of the control structure, the assignment must be in the program parts P", R_n, or Q.

Case a

The assignment is in P" and therefore of the form $\alpha \leftarrow def(\alpha)$. All qualified names β in $def(\alpha)$ fall into one of three categories:

(1) β = Y.j. Then β is not assigned in P", R_n or Q, hence, by part 1, $v(\beta) = (|Y:j|_\sigma)_0 = (|Y:j|_\sigma)_r$ (def. of :i).

(2) β = X[i] and X[i] is assigned in P" by a statement preceding the assignment to α. By hypothesis, $v(\beta) = (|h(\beta)|_\sigma)_r$

(3) β = X[i] and X[i] was assigned last in Q. By hypothesis, as t has not been assigned since, $v(\beta) = (|h(\beta)|_\sigma)_r$.

Therefore, for all qualified names β in def(α),

$$v(\beta) = (|h(\beta)|_\sigma)_r.$$

Observe next, that by construction of L_n by A4.1, $\alpha \neq Z[0]$ for some Z defined by a fby expression. Hence, by Theorem 3.1, for all solutions ρ,

$$(|h(\alpha)|_\rho)_r = (|h(def(\alpha))|_\rho)_r$$

for all values of r. Consequently, in particular

$$(|h(\alpha)|_\sigma)_r = (|h(def(\alpha))|_\sigma)_r$$

from which, and the above observations, follows

$$v(\alpha) = (|h(\alpha)|_\sigma)_r .$$

Case b

The assignment is of the form R.0 ← E[n]. Note that the assignment is executed for the first time, but F[n] is tested for the r-th time. Consequently, by hypothesis,

$$v(F[n]) = (|F\langle n\rangle|_\sigma)_r = (|F|_\sigma)_{r+n} = \underline{\text{true}},$$

$$(|F\langle n\rangle|_\sigma)_s = (|F|_\sigma)_{s+n} = \underline{\text{false}} \quad \text{for all } s < r, \text{ and}$$

$$(|F\langle k\rangle|_\sigma)_0 = (|F|_\sigma)_k = \underline{\text{false}} \quad \text{for all } k < n.$$

By definition of asa and by hypothesis, therefore,

$$v(R.0) = v(E[n]) = (|E\langle n\rangle|_\sigma)_r = (|E|_\sigma)_{r+n} = (|R:0|_\sigma)_s$$

for all values of s.

Case c

The assignment is in Q and of the form X[i] ← X[i+1]. By reasoning analogously to Case c of Part 1 of the proof,

$$v(X[i]) = (|X\langle i\rangle|_\sigma)_r$$

□

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 $(|F|_\sigma)_s$ is false for all s < t, then $(|R|_\sigma)_{t'}$ is undefined, 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 of the result follows from the previous theorem. Note, however, that because of the if - then - else, some sequence components may have been evaluated, which are not used in some particular iteration.

□

5. Compilation of Simple Programs

Even simple programs will, in general, contain more than one variable defined by an asa expression. A suitable modification of the algorithms of the previous section is in order, so that they can be used to compile the more general case.

From the dependency graph G of a block 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'.

Definition The graph G' derived from the dependency graph G as described above is the goal graph.

Theorem 5.1 The goal graph of a block in a subset program does not contain any cycles.

Proof There cannot be a cycle in G' passing through the node result, since there are no edges into the corresponding node of G. Suppose next that there is a cycle in G' passing through node X. By the construction of G', therefore, there is also a cycle in G passing through the node X of G. Since all arcs emanating from X in G are labelled ∞, the edge sum of this cycle in G cannot be negative, contrary to subset requirements.

□

The modifications to the algorithms do the following. 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 the quiescence of goals, each such procedure needs to be called at most once, since the value cannot change subsequently. Efficient code taking advantage of this property is easily generated, but only sketched in its structure. Essentially it involves testing a special variable which indicates whether or not the goal whose value is needed has ever been evaluated. The code evaluating result then acts as the main-line program. Note, however, that the procedures may call each other as well. Because of Theorem 5.1, no recursive calls are ever made.

The modifications of A4.1 and A4.4 which accomplish this task are described below. The modified algorithms are referred to as A5.1 and A5.4, respectively.

Modification of A4.1

Replace Step 3 of the algorithm by

3' [Process and Extend L_n. Construct Dependency Lists]

Take the next item α in L_n and initialize its dependency list D to be empty. If α is not derived from any goal, then process def(α) as described below. For each β in def (α) do the following:

(i) If β is derived from some goal Z, then replace β by Z.0.

(ii) If β is not in $L_0,\ldots,L_n$, then append β to L_n.

(iii) If β is not in D, then append β to D.

Theorem 5.2 Algorithm A5.1 terminates for arbitrary goals R = E asa F in simple programs in the subset.

Proof Obvious.

Modification of A4.4

Replace Steps 1 and 2 in A4.4 by the corresponding steps below.

1' [Initialize and Code S]

If R is result then

emit "begin INIT;

t ← 0; ";

otherwise

emit " procedure R begin

SET_R;

t ← 0; ",

where INIT is code initializing all the special variables to indicate that none of the goals in the simple program have been evaluated, and SET_R is code assigning a special variable to indicate that the goal R has been evaluated.

Scan S and code each item α as prescribed by Step 2'. Thereafter, go to Step 3.

2' [Code item α in S]

If α is a qualified name derived from a goal Y, then

emit " $TEST_Y$; ",

where $TEST_Y$ is code testing a special variable whether the goal Y

has been evaluated, and if it has not been, calls procedure Y.

If α is a qualified name not derived from a goal, then

emit " $\alpha \leftarrow \beta$; ",

where β is the defining term of α modified by replacing each qualified name γ which is derived from a goal Y in it by Y.0.

If α is $\#_j$, then

```
emit " if F[j] then begin
                        R.0 ← E[j];
                        goto L;
                        end;
```

If α is %, then

emit " repeat forever begin "

Note that for each goal Z, first Z = Z, which justifies the modification of the defining terms by A5.4. The label L generated by A5.4 is local to the section of code for the particular goal, and likewise the allocated storage for t and all qualified names. Exceptions to this are the names R.0, where R is a goal, and the special variables which indicate whether or not a goal has been evaluated.

Theorem 5.3 The code π compiled for a goal R = E asa F in a simple program in the subset evaluates R correctly, provided the correct value of each goal Y referenced in π is available in Y.0.

Proof (Informal)

Since the goal graph is cycle free, observe that π cannot reference R.0. If R is a simple goal, no other goals are referenced, and the modifications to the algorithms do not affect the code compiled to evaluate R. Correctness in this case follows from Corollary 4.9.

Otherwise, since each goal Y is quiescent, the modified defining term β of α has the same interpretation as $def(\alpha)$. A proof analogous to the proofs of Theorem 4.8 and Corollary 4.9 therefore establishes the Theorem.

□

Whenever result is not a goal, it must be defined by

result = E,

where E is quiescent, hence its compilation can be reduced to the compilation of goals by transforming it into

result = E asa true.

This results in a slight loss of efficiency. A simple modification anticipating the former definition as a special case is worked out easily and compiles better code. The details are left to the reader.

Now the algorithm for compiling simple programs can be given.

Algorithm A5.0 (Compilation of Simple Programs)

Input: A simple program P in the subset

Output: Object program π evaluating output.

1. [Find all goals]

   Construct the goal graph for P and reduce it by deleting all nodes which cannot be reached from result.

2. [Linearize]

   Linearize the reduced goal graph by a topological sort. (Note that the sort must succeed because of Theorem 5.1).

3. [Compile]

   Compile each goal separately in the sequence of the linearized graph by Algorithms A5.1, A4.2, A4.3, and A5.4.

4. [Complete Object Program]

   Prefix the concatenated program parts by " begin " and suffix them by "output [0] ← result.0; print(output[0]); end;"

Theorem 5.4 The program π compiled by A5.0 for a simple program P in the subset evaluates each goal referenced correctly, and does not reference an unevaluated goal.

Proof By Theorem 5.1, the goal graph does not contain any cycles, and hence no recursive calls are made on the routines evaluating goals. Also, before referencing any goal Y, the special variable T_Y, indicating whether Y has been evaluated, is tested. Since the main program initializes all such special variables correctly, and only the procedure

evaluating Y can assign T_Y subsequently, Y is evaluated before it is referenced for the first time.

The rest follows from Theorem 5.3 and an induction on the goal graph.

□

Corollary 5.5 (Partial Compiler Correctness)
Let π be the program compiled by Algorithm A5.0 for the simple program P in the subset, and let σ be the minimal fix-point 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 observing that A5.0 appends the concatenated program parts with "output[0] ← result.0; print(output[0]); end. "

If the value $(|\text{output}|_\sigma)_0$ 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 is accomplished in the presence of more than one asg expression. This is due to the simplified treatment of the if - then - else, and discussed further in Section 7.

6. Nested Blocks

Two properties of nested blocks suggest a recursive extension of the compilation algorithms for simple programs, which enables compiling programs with nested blocks.

In a nested block, all global variables referenced are quiescent, by the definition of the language, and therefore only the current component values of global variables need to be known throughout the evaluation of a given nested block. Thus, the global environment 'freezes' upon entry into a nested block.

Furthermore, if block B_2 is nested within block B_1, and $G^{(1)}$, ... $G^{(j)}$ are all variables in the using clause of B_2, then the evaluation of the sequence R defined by B_2 may be considered a point-wise function with arguments $G^{(1)}$ ... $G^{(j)}$.

Theorem 6.1 Let R be defined by the block B_2 nested in a block B_1, where B_2 is

compute R using $G^{(1)}$... $G^{(j)}$ where ... end.

Then R is a point-wise function f in B_1 with arguments $G^{(1)}$... $G^{(j)}$,

i.e. $R = f(G^{(1)}, .., G^{(j)})$,

first $R = f(\text{first } G^{(1)}, .., \text{first } G^{(j)})$,

next $R = f(\text{next } G^{(1)}, .., \text{next } G^{(j)})$.

Proof (Sketch; for a fully detailed proof see [5].)

The proof proceeds by induction on the block structure of the program.

Basis Assume that B_2 does not contain any nested blocks. Let $G^{(1)}$ ... $G^{(r)}$ be all variables globally referenced in B_2 (-- B_2 may reference variables global to B_1 as well), and observe that implicitly

one or more latest are applied to the $G^{(k)}$. Let

$$\bar{G}^{(k)} = \underset{\sim}{\text{latest}}^{i_k}\ G^{(k)}, \quad 1 \le k \le r, \quad i_k > 0.$$

The interpretation of every expression H in B_2 may be considered a function f_H of the $\bar{G}^{(k)}$. Let $\bar{t} = t_0 t_1 t_2 \ldots \in N^N$, then

$$(|H|)_{\bar{t}} = (f_H)_{\bar{t}}\ (\ (|\bar{G}^{(1)}|)_{\bar{t}}\ ..,\ (|\bar{G}^{(r)}|)_{\bar{t}})$$

Since latest and $\underset{\sim}{\text{latest}}^{-1}$ are the only operations manipulating the $t_1 t_2 \ldots$, and since all of these are applied to the $G^{(k)}$, f_H does not depend on the $t_1 t_2 \ldots$ and varies with t_0 only:

$$(f_H)_{\bar{t}} = (f_H)_{t_0}$$

In particular, because of its quiescence, for the result expression E,

$$(f_E)_{\bar{t}} = (f_E)_{t_0} = (f_E)_0 = f_E$$

Therefore, in B_1,

$$\begin{aligned}(|R|)_{t_1 t_2 \ldots} &= (|\underset{\sim}{\text{latest}}^{-1}\ (f_E(\bar{G}^{(1)} \ldots \bar{G}^{(r)}))|)_{t_1 t_2 \ldots} \\ &= (|f_E(\bar{G}^{(1)} \ldots \bar{G}^{(r)})|)_{0 t_1 t_2 \ldots} \\ &= f_E\ (\ (|\bar{G}^{(1)}|)_{0 t_1 t_2 ..}, \ldots, (|\bar{G}^{(r)}|)_{0 t_1 t_2 ..})\end{aligned}$$

Let $\tilde{G}^{(k)} = \underset{\sim}{\text{latest}}\ \bar{G}^{(k)}$, then

$$(|R|)_{t_1 t_2 \ldots} = f_E\ (\ (|\tilde{G}^{(1)}|)_{t_1 t_2 \ldots}\ , \ \ldots, \ (|\tilde{G}^{(r)}|)_{t_1 t_2 \ldots}\ )$$

By assumption, the $G^{(1)} \ldots G^{(j)}$ are all defined in B_1, hence $\tilde{G}^{(k)} = G^{(k)}$, $k \le j$, and at least one latest is applied to the $\tilde{G}^{(j+1)} \ldots \tilde{G}^{(r)}$, hence $(|\tilde{G}^{(k)}|)_{t_1 t_2 \ldots} = (|\tilde{G}^{(k)}|)_{0 t_2 \ldots}$, $k > j$, from which the theorem follows.

Induction Step Analogous in argument. Observe, however, that additional global references may introduce further arguments to the functions representing the interpretation of the result expression of nested blocks, on which the functions depend trivially.

□

The theorem justifies the conceptualization of the evaluation of R in B_1 as a point-wise formula with operands $G^{(1)} \ldots G^{(j)}$, and therefore a recursion on the block structure is an adequate compilation strategy. Once B_2 is compiled, it is viewed as a subprogram evaluating R, which is called at the appropriate point in the code for B_1.

Corollary 6.2 If R is defined by a block B_2 nested in a block B_1, and the using clause of B_2 is either empty or contains only quiescent names, then R is quiescent in B_1.

Proof Obvious.

This result explains the definition of quiescence in Section 2.4. It can also be used for a code transformation which, upon recognizing the quiescence of R, "lifts" the nested loop evaluating R to the containing loop level transforming it into a concatenated loop. This is accomplished by replacing every reference to R in B_1 by first R. This technique is well-known from compiler code optimization as moving invariant computations out of loops.

Definition Let R be defined by

compute R using $G^{(1)}$ ... $G^{(j)}$ where ... end.

Then R[i] depends directly on $G^{(k)}$[i], and R.i depends directly on $G^{(k)}$.i, $k \leq j$.

This extension of the notion of dependency is justified by Theorem 6.1. Note that the theorems of Section 3 generalize immediately. This, and the results of Section 4 and 5 are the basis for proving the correctness of the modified algorithms described below. Instead of giving all the details, we outline the algorithms intuitively.

Algorithm A5.0 is adapted by making it recursive and capable of compiling R[i] or R.i, where R is defined by a nested block. Initially called to compile the outermost block for output[0], it proceeds as follows.

Compiling R[i] or R.i, R defined by a block B, the modified A5.0 compiles B as if it were a simple program which, in addition, requires certain global scalar values. Names derived from a global variable G are neither added to the operand lists, nor to the dependency lists. Instead, they are replaced by *G and collected into a separate list, the parameter list of B. Since G is quiescent in B, all first and next applied to it can be dropped without altering the meaning of B.

Recursive calls on A5.0 are made by the modified A5.1 for blocks nested in B.

Upon completing the compilation of B for R[i] or R.i, the

parameter list of B is examined. Any sequence G in the list, which is defined in the containing block (and is therefore in the using clause of B) is an operand of the pointwise computation of R, and hence references the qualified name G[i] or G.i, respectively. All other names in the parameter list are global to the containing block, and hence to be added to its parameter list.

The modified algorithm A5.1 proceeds as follows. When examining R[i] or R.i, where R is defined by a block B, the dependency list of R[i] or R.i, respectively, is set to contain the name $G^{(k)}$[i] or $G^{(k)}$.i, respectively, where the $G^{(k)}$ are the names in the using clause.of B. Then a recursive call is made to compile B for R[i] or R.i. Note that B needs to be compiled only once, since the compilations for different qualified names differ only in the bindings of the global references of B. The compiled code is temporarily stacked.

Algorithms A4.2 and A4.3 remain unchanged. A5.4 is modified such that, when coding R[i] or R.i, R defined by block B, the compiled code of B is inserted, and the global references to $G^{(k)}$ are resolved by binding them to $G^{(k)}$[i] or $G^{(k)}$.i. Note that B has local storage for all names defined within B.

From the reference graph requirements and from Theorem 4.1 it should be clear that no infinite recursion is possible, and that the compilation always terminates. Arguments analogous to those of Sections 4 and 5 establish the following

Theorem 6.3 Let σ be the minimal solution of the program P in the subset, and assume that P contains the definition

compute R using $G^{(1)}$ ... $G^{(j)}$ where ... end.

If, upon entry into the code π compiled for R[i], we have

$$v(G^{(k)}[i]) = (|G^{(k)}\langle i\rangle|_\sigma)_{\bar{t}}$$

as the current value of the $G^{(k)}[i]$, $1 \le k \le j$, then, if π terminates, the value of R[i] upon exit from π is

$$v(R[i]) = (|R\langle i\rangle|_\sigma)_{\bar{t}}$$

and, if

$$(|R\langle i\rangle|_\sigma)_{\bar{t}} = \perp,$$

then π does not terminate.

Analogous formulation and proof for R.i establishes the (partial) correctness of the compiler.

We conclude the section with the compilation of the second program of Example 2.1. For reasons explained in the next section, the compiled code is correct, rather than only partially correct.

Example 6.1

Consider the second program of Example 2.1. The compiler is called to compile the outermost block for output[0]. Since the goal graph has only one node, the modified A5.1 compiles result and constructs the following lists before recursively calling the compiler for the inner block:

	operand lists	dependencies	parameter list
L_0:	t1[0]	sum[0]	--
	p1[0]	c[0], m[0]	
	sum[0]	--	
	c[0]	--	
	m[0]	input.0	
	input.0	--	
L_1:	t1[1]	sum[1]	
	p1[1]	c[1], m[1]	
	sum[1]	sum[0], root[0]	
	c[1]	c[0]	
	m[1]	input.0	
	root[0]	n[0]	
	⋮		

When processing root[0], a recursive call is made to compile the inner block for root[0]. The inner block, too, has a goal graph consisting of a single node, and constructs the following lists:

	operand lists	dependencies	parameter list
L_0:	t2[0]	cc[0]	*n
	p2[0]	y[0]	
	cc[0]	--	
	y[0]	--	
L_1:	t2[1]	cc[1]	
	p2[1]	y[1]	
	⋮		

From these tables, the following code is compiled for the inner block:

```
begin   t' ← 0;
        cc[0] ← 0;
        t2[0] ← cc[0];
         y[0] ← 1;
        p2[0] ← y[0]  gt  *n;
        if  p2[0]  then begin result' ← t2[0];
                                goto L';  end;
        repeat forever begin
                        cc[1] ← cc[0]+1;
                        t2[1] ← cc[1];
                         y[1] ← y[0]+2*cc[0]+3;
                        p2[1] ← y[1]  gt  *n;
                        if  p2[1]  then begin result' ← t2[1];
                                                goto L';  end;
                           t' ← t'+1;
                        cc[0] ← cc[1];
                         y[0] ← y[1];
                        end;
L':  *root ← result'.0;

end;
```

After returning to the calling level, *n is found to reference n in the outer block, hence the parameter list of the outer block remains empty. Processing resumes with n[0] and, after completing, the following code is compiled for the outer block:

```
begin   t ← 0;
 input.0  ← read();
   sum[0] ← 0;
    t1[0] ← sum[0];
     c[0] ← 0;
     m[0] ← input.0;
    p1[0] ← c[0]  eq  m[0];
    if  p1[0]  then begin  result.0 ← t1[0];
                           goto L;    end;
    repeat forever begin
                  c[1] ← c[0]+1;
                  m[1] ← input.0;
                 p1[1] ← c[1]  eq  m[1];
               input[1] ← read();
                  n[0] ← input[1];
                  begin
                        (code compiled for nested block
                     :   substituting n[0] for *n, and
                     :   root[0] for *root)
                  end;
                sum[1] ← sum[0]+root[0];
                 t1[1] ← sum[1];
                 if  p1[1]  then begin  result.0 ← t1[1];
                                        goto L;    end;
                    t ← t+1;
                sum[0] ← sum[1];
                  c[0] ← c[1];
                  end;
L:  output[0] ← result.0;
    end;
print(output[0]);
```

The read function will in general have two parameters, one to indicate the index of the qualified name derived from input, and one to differentiate input[i] from input.i. In the example this was omitted, because the control structure ensures that the correct values are read.

7. Practical Considerations and Compiling Problems

7.1 Syntactic Correctness

The compiler as designed in the previous sections relies on syntactically correct input, and clearly any implementation must perform syntax diagnostics as well. Much of this can be done by the familiar methods, except for Criterion 2.5.5, a direct verification of which seems hardly practical. Recall how this property, that the edge label sum of any cycle in the dependency graph be negative, influences the compilation: It guarantees termination of Algorithms A4.1 and A4.2, implies that the goal graph of any block is cycle-free, and that no infinite recursion for compiling nested blocks is possible. This suggests to add appropriate routines throughout the compilation which test for the presence of these situations.

It is possible to prove that if there is a cycle in the dependency graph with non-negative edge label sum, then Algorithm A4.1 or A4.2 does not terminate, or the goal graph cannot be linearized, or there is an infinite recursion on the block structure. In particular, using Theorem 3.3, the following can be proved:

Theorem If the list L_i constructed by Algorithm A4.1 (or A5.1) contains a qualified name X[j] or X.j such that $j > s+i+1$, where s is the number of next operators occurring in the block presently compiled, then the list L_i cannot be finite, and there is a cycle in the reference graph with edge label sum greater than 0.

Consequently it is possible to test whether A4.1 terminates, and if it does not, then the syntax is violated. Termination of the topological sorts A4.2 and of the goal graph is easily tested, and the presence of an infinite recursion on the block structure can be known when more than m calls are made on the compiler, where m is the total number of blocks. This gives a practical way of testing Criterion 2.5.5.

The details of how to obtain the information necessary to give meaningful diagnostics in each of the above situations are left to the reader.

7.2 Object Code Efficiency

As the examples of compiled code indicate, the generated object code is somewhat lengthy. Its simple control structure makes it particularly gratifying to adapt conventional code optimization methods [see, for example, 1] to improve the generated code. Simple strategies can be worked out which would reduce the program of Example 4.5 to less than half its size eliminating two tests and several qualified names.

7.3 Logical Connectives

In the original language definition [4, Section 3.1] we find the following interpretation of the operator or:

$\vee_S$ yields *true* if at least one argument is *true*, *false* if both are *false*, *undefined* otherwise.

Thus, in particular,

true $\vee_S$ $\perp$ = $\perp$ $\vee_S$ *true* = *true*

Since $\perp$ cannot be tested, the operands of *and* and *or* should be evaluated in parallel, which, in conjunction with other language constructs, implies spawning an arbitrary number of parallel processes at run time, some of which have to be aborted once other ones terminate with certain values. Efficient compilation of this seems very difficult, and the two operators have been made strict in the language definition on which the compiler is based.

7.4 Partial Correctness

The algorithms developed accomplish only partial correctness. This is due entirely to the treatment of conditional expressions, and is perhaps best appreciated through an example.

Consider the following section of code:

```
Z = if P then X else Y;
X = X0 fby f(X,G);
G = E asa F;
```

where f(X,G) represents some computation referencing X and G. Assume, that Z is to be evaluated by iteration, and that the evaluation of Y does not need the value of G.

Because of the recursiveness of X, it is of advantage to evaluate

it by an iteration, and the present algorithms accomplish this and fuse the loops for X and for Z. However, if P is always false, then X need not be evaluated, and since X references G, partial correctness is introduced if, in that situation, the evaluation of G would not terminate. If, on the other hand, the evaluation of Y references G also, then the loop fusion does not introduce partial correctness and results in superior code.

Further analysis reveals that, in general, the object code must post-pone sequence evaluations until they are required, and even avoid evaluating some components altogether. These considerations lead naturally to an interpretive, demand-driven implementation [see also 6], and the two existing interpreters for Lucid have taken this approach. However, present research indicates that, at the expense of considerably more sophisticated dependency analysis and object code structure which is not interpretive, full correctness is feasable. It involves, on the object code level, a mixture of iterative co-routines and recursive procedures constituting units which evaluate various sequences, not all of which will be goals. This can be done since most recursions in source programs can be replaced by a collection of iterative routines. All recursion, however, cannot be eliminated, for example for the following variable

$$X = X_0 \text{ fby } (\text{if } P \text{ then } f(X) \text{ else } Y);$$

it is not possible.

7.5 Subset Choice

The fundamental choice of our approach to compiling the language was to interpret the index sequence $\bar{t}$ as a sequence of time parameters. Consequently, the subset was defined to permit an efficient iterative evaluation of programs requiring only a fixed amount of storage predictable at compile time. In this frame, certain constructs are therefore unnatural, even though they make sense in the fix-point semantics. For example, the definition of a variable in terms of its 'future' is possible:

X = if P then Y else next X.

The crucial restriction is Criterion 2.5.5, which eliminates all such constructions, even though some may have equivalent formulations which satisfy the criterion (for example, X above could be written as X = Y asa P).

It is not clear how a larger compilable subset can be defined without having to resort in part to interpretation. It seems, however, that major changes to the algorithms would have to be made in an attempt to accomplish this.

8. Conclusions

Several 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, although better ones would be desirable, 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 still not available. Perhaps this task is easier for data-flow languages and this may be another reason for the recent interest in them. On the other hand, it is not clear how such languages are to be factored into orthogonal constructs given the presently available target machine architectures, so that a correctness proof and a compiler design may be modularized in a manner such, that as proposed by Morris in [17]. In the case of Lucid the problem was to some extent solved by recognizing the key role played by the asa operator.

Furthermore, it is felt that the style of analysis performed by the compiler is applicable to a broader class of languages and, therefore, is of interest in itself. The influence which different machine archtectures, e.g. network machines, would have on the designs seems particularly promising for future research. The features of the source language itself, finally, deserve attention. It is encouraging that a language chiefly motivated by program proving should share essential properties in design with data-flow

languages which are motivated by the study of parallelism. Thus it appears, that the work will stimulate deeper insights into these problems.

Acknowledgements

The many helpful discussions with, and the constructive criticism of, Ed Ashcroft, Andy Blikle, and Tom Maibaum are gratefully acknowledged.

References

1. Allen, F.E.

Program Optimization
in Annl. Review of Autom. Programming Vol. 5, p. 239-308
Pergamon, N.Y. 1969.

2. Ashcroft, E.A.

Program Proving without Tears
Proc. of the Intl. Symp. on Proving and Improving Programs,
p. 99-111, Senans, France, July 1975

3. Ashcroft, E.A., and W. Wadge

Lucid, a Non-Procedural Language with Iteration
forthcoming in Comm. of the ACM, 1976
also Tech. Report CS-75-02, Dept. of Comp. Science,
University of Waterloo (22 p.)

4. Ashcroft, E.A., and W. Wadge

Lucid, A Formal System for Writing and Proving Programs
forthcoming in SIAM Journal of Computing, Sept. 76

5. Ashcroft, E.A., and W. Wadge

Advanced Lucid
Technical Report CS-76-22, Dept. of Computer Science
University of Waterloo, 1976 (forthcoming)

6. Cargill, T.

Deterministic Operational Semantics for Lucid
Technical Report CS-76-19, Dept. of Computer Science,
University of Waterloo, 1976.

7. Chirica, L.M., and D.F. Martin

An Approach to Compiler Correctness
Intl. Conf. on Reliable Software, p. 96-103,
June 1976 also SIGPLAN Notices Vol 10, No. 6

8. Dennis, J.B.

First Version of a Data-Flow Procedure Language
MIT Project MAC Tech. Memo No. 61, May 1975

9. Floyd, R.W.

Assigning Meanings to Programs
Mathematical Aspects of Computer Science,
Vol. 19, p. 19-32
Providence, R.I. 1967

10. Guttag, J.

Abstract Data Types and the Development of Data
Structures Suppl. to Proc. of the ACM Conf. on
Data, p. 37-46, Salt Lake City, Utah, March
1976

11. Hoare, C.A.R., and N. Wirth

An Axiomatic Definition of the Programming
Language PASCAL Acta Informatica Vol. 2,
p. 335-355, 1973.

12. Kosinski, P.R.

A Data Flow Programming Language
IBM Research Rep. RC-4264, 134 p., March 1973

13. London, R.L.

Correctness of Two Compilers for a LISP Subset
A.I. Memo 151, Stanford University, 1971

14. McCarthy, J., and J.A. Painter

Correctness of a Compiler for Arithmetic Expressions
Math. Aspects of Computer Science, Vol. 19,
Amer. Math. Soc., Providence, R.I. 1967

15. Miller, R.E., and J. Cocke

Configurable Computers: A New Class of General Purpose Machines, IBM Research Rep. RC 3897, 14 p. June 1972

16. Milner, R., and R. Weyhrauch

Proving Compiler Correctness in a Mechanized Logic
Machine Intelligence 7, p. 51-71, Edinburgh University 1973

17. Morris, F.L.

Advice on Structuring Compilers and Proving Them Correct, ACM Symp. on Principles of Prog. Lang., p. 144-152, Boston, Mass. Oct. 1973.

18. Rumbaugh, J.E.

A Parallel Asynchronous Computer Architecture for Data-Flow Languages
MIT Project MAC Rep. TR-150, 319 p., May 1975

19. Van Emden, M.

Verification Conditions as Representations for Programs
Research Report CS-76-03, Dept. of Comp. Sci., University of Waterloo 21 p., 1976