[illegible]

WORK FILE } MASTER

TECHNICAL REPORT

p47!

TR 25.110
12 August 1970

PROVING CORRECTNESS OF IMPLEMENTATION TECHNIQUES

C. B. JONES
P. LUCAS

IBM LABORATORY VIENNA

IBM LABORATORY VIENNA, Austria

PROVING CORRECTNESS OF IMPLEMENTATION TECHNIQUES

C.B. JONES
P. LUCAS

ABSTRACT:
This report illustrates how proofs of the correctness of compiler techniques can be based on a formal definition of the language being implemented. A detailed proof of the display method of referencing variables is given as an example.

Locator Terms for IBM Subject Index
Compilers
Programming languages
Correctness
Proof
Block structure
Semantics
21 PROGRAMMING

TR 25.110
12 August 1970

CONTENTS

1. INTRODUCTION

The existence of formal definitions of programming languages invites attempts to find a more systematic approach to the design of implementations. In particular, the work described below adopts the approach of isolating common parts of programming languages, modelling possible implementations on different levels of abstraction and exhibiting the correctness of such implementations.

The so-called Vienna Method is a collection of techniques based on abstract machines which interpret programs. These techniques are also used to define the implementation methods. The proof that an appropriately defined equivalence relation holds between the defining and implementation machines establishes the correctness of the latter.

From the subjects so far investigated the block concept has been chosen to provide the example for this paper. The implementation method considered is the well-known display method.

All notation used is introduced in the next chapter, thus making the paper self-contained.

2. METHOD AND NOTATION

An attempt has been made to minimize the use of unconventional notation. However, some of the notation used in the Vienna definition work proved to be useful even for the simple example presented in this paper. To make this paper self-contained this notation is introduced in section 2.1 below, as far as necessary, a more detailed exposition can be found in /Lucas, Walk 1969/.

2.1 Objects

Objects are either composite or elementary. A set of elementary objects is presupposed, natural numbers and certain sets of names are examples of elementary objects used in this paper (see section 3.1.2). Composite objects can be considered as finite tree structures with elementary objects at the terminal nodes. Branches are named with the restriction that two branches starting from the same node must not have the same name. Names of branches are called selectors. Subtrees of a given object are called components. Components directly attached to the root of a given tree structure are called immediate components.

The composite object with zero components is called the null object and is denoted by Ω.

Selectors are used to select components from given objects. Let s be a selector and x an object then:

s(x) denotes the immediate component of x whose name is s.

If there is no such component, including the case that x is an elementary object, then $s(x) = \Omega$.

The selection of components may be iterated, e.g. s1(s2(x)) denotes the s1 component of the s2 component of x.

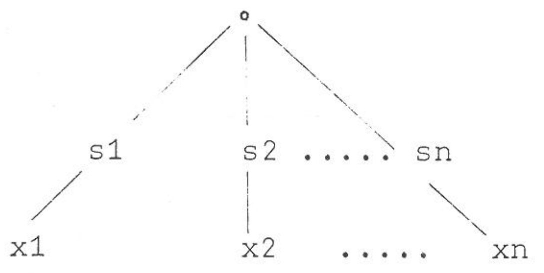
Predicates over objects are defined in an obvious notation (see /McCarthy 1962/). An object is said to be of a certain type if it satisfies a corresponding predicate.

In the sequel some objects will be used to represent mappings from certain finite sets into finite sets. By analogy with the terms used for mappings one may talk about the domain D(x) and the range R(x) of a given object, where:

$$D(x) \underset{Df}{=} \{ s \mid s(x) \neq \Omega \}$$
$$R(x) \underset{Df}{=} \{ s(x) \mid s(x) \neq \Omega \}.$$

For example:

let x =



$D(x) = \{s1, s2, \ldots, sn\}$, $R(x) = \{x1, x2, \ldots, xn\}$

As a consequence: $R(\Omega) = D(\Omega) = \{\}$

To remain in the present framework of objects, lists can be considered as objects whose immediate components are named by a subset of the natural numbers. However, in this paper elements of lists are referred to by the conventional subscript notation.

The following conventions are adopted for non-empty lists L:

let $L = \langle L_n, L_{n-1}, \ldots, L_1 \rangle$

$l(L) \underset{Df}{=} n$ length of the list

$top(L) \underset{Df}{=} L_n$

$rest(L) \underset{Df}{=} \langle L_{n-1}, \ldots, L_1 \rangle$

for $1 \leq i \leq n$:

$rest(L,i) \underset{Df}{=} \langle L_i, L_{i-1}, \ldots, L_1 \rangle$

The following consequence is used:

(N1) for $L1 = rest(L2)$ & $1 \leq i \leq l(L1)$:

$$rest(L1,i) = rest(L2,i)$$

2.2 Abstract Machines

For the present purpose an abstract machine is defined by a set of initial states and a state transition function, $f(tx,\xi)$, which maps program texts, tx, and states into states.

A computation for a given initial state ξ^1 and a given program text tx is a sequence of states:

$$\xi^1, \xi^2, \ldots, \xi^i, \xi^{i+1}, \ldots \quad \text{such that } \xi^{i+1} = f(tx,\xi^i)$$

The set of all possible states which the machine can assume is the set of all states occuring in computations of arbitrary initial states and program texts.

The state transitions of the underlying machine for chapter 3 will not be defined explicitly by a function but will only be constrained by certain postulates. This is because the paper is not about a specific language but a common part of many languages.

2.3 Twin Machines

The definitions of both the language part and its implementation are specified by abstract machines. For the purpose of proving certain equivalences between such machines /Lucas 1968/ introduced the approach of combining the two machines into one, a so-called twin machine . Combining two machines into a twin machine means assembling the state components of the two machines into state components of the twin machine and defining the state transitions accordingly. The equivalence problem can then be formulated as a property of the states of the twin machine.

2.4 Notation Summary

<u>logic</u>:

$\neg$	not
&	and
$\vee$	or
$\supset$	implication
$\forall$	universal quantifier
$\exists$	existential quantifier

<u>sets</u>:

$\in$	element of
$\notin$	not element of
$\cup$	union
$\cap$	intersection
$\subseteq$	subset or equal
$\{\}$	empty set

<u>conditional expressions</u>:

(prop $\longrightarrow$ expr1,expr2)	if prop then expr1 else expr2

<u>objects</u>:

s(x)	s component of x for: s is a selector x is an object
I	identity selector (I(x) = x)
Ω	null object

R(x)	range of x (Definition $\{s(x) \mid s(x) \neq \Omega\}$)
D(x)	domain of x (Definition $\{s \mid s(x) \neq \Omega\}$)
tp1 $\Leftarrow$ tp2	tp1 immediately contained in tp2 (Definition see section 3.1.1)

<u>naming convention:</u>

s- ...	selector name
is- ...	predicate name

<u>lists</u>: let L = $<L_n, L_{n-1}, \dots L_1>$

l(L)	length(=n)
top(L)	L_n
rest(L)	$<L_{n-1}, \dots, L_1>$
rest(L,i) =	$<L_i, L_{i-1}, \dots, L_1>$

3. DEFINITION AND IMPLEMENTATION OF THE BLOCK CONCEPT

This chapter presents a definition of the block concept as first introduced into programming languages by ALGOL 60 /Naur et al. 1962/ and a proof of correctness of a formulation of the stack mechanism used in the first complete implementations /Dijkstra 1960/.[1)]

The programming language under consideration is assumed to include the concept of blocks and procedures, and the possibility to pass procedures as arguments. Go to statements and general call by name are not considered in order to avoid burdening the proofs.

The definition of the block concept is derived from the formal definitions of PL/I and ALGOL 60 /Walk et al. 1969/, /Lauer 1968/. However, for this paper the structure of programs being interpreted as well as the state and state-transitions of the abstract machine are specified only as far as is necessary to establish the subsequent proofs. This definition therefore illustrates the essentials of the block concept and the assumptions about the language necessary to guarantee the applicability of the stack mechanism.

The stack mechanism itself is also formulated as an abstract machine which,in the opinion of the authors, expresses the essential properties of this mechanism. No attempt is made to make the additional step from the general result to an actual machine implementation.

The proof of correctness is based on the twin machine, specified in section 3.1, which combines the defining machine and the stack mechanism.

---

1) see also /Van der Mey 1962/.

The emphasis of the chapter is on the technique used to specify the block concept, its implementation and on the proof of correctness. The end result has been known for many years but, apart from being an example of formally proving implementations correct, the proof indicates an argumentation which could be used informally in lecturing on the subject matter.

3.1 The Defining Model and its Implementation

3.1.1 The structure of programs

For the purpose of the present paper it is sufficient to state that programs form a completely nested structure of blocks and procedure bodies. A program is itself a block which is referred to as the outermost block. More precisely, programs are objects of type is-block containing possibly nested components of type is-block or is-body.

Selectors [1] to components of programs are called text pointers (tp). Consider a program, tx, and the set of all textpointers to blocks or procedure bodies of that program BP_{tx}. The text pointer to the entire program is I and $I \in BP_{tx}$ since programs are blocks.

The relation of the block pointed to by tp1 being immediately contained in the block pointed to by tp2 is written:

$$tp1 \Longleftarrow tp2 \qquad \text{for } tp1, tp2 \in BP_{tx}$$

1) Note that these are composite selectors, i.e. any selectors s1, s2,..., sn may be combined to form a composite selector s1∘s2∘...∘sn where:

$$s1\circ s2\circ\ldots\circ sn(x) \underset{Df}{=} s1(s2(\ldots(sn(x))\ldots))$$

Since blocks and procedure bodies are completely nested, each tp1, $tp1 \in BP_{tx}$ & $tp1 \neq I$, has a unique tp2 such that $tp2 \in BP_{tx}$ and $tp1 \Longleftarrow tp2$. Thus, for any given tp1, $tp1 \in BP_{tx}$, there is a unique chain of predecessors ending with I:

$$tp1 \Longleftarrow tp2 \Longleftarrow \ldots \Longleftarrow I$$

For each block or procedure body there is a set of identifiers:

ID(tx,tp) for is-block(tp(tx)) $\vee$ is-body(tp(tx))

called the set of locally declared identifiers.

It is not necessary for the present purpose to state the attributes which can be declared with the identifiers nor how these attributes would be associated with the identifiers.

A pair (id,tp) is called an <u>occurence</u> of the identifier id in tx if tp(tx) = id.

An occurence is said to be immediately contained in a block or procedure body, tp1, if its pointer part is contained in tp1 but is not contained in any block or procedure contained in tp1.

3.1.2 States of the twin machine and their significance

The set of states of the twin machine (i.e. defining model) is the set of objects generated from the set of initial states by the state transitions(both initial states and state transitions are characterized in section 3.1.3). It would not be necessary to characterize the states in any other way. However, it seems advantageous to give a layout of the structure of the states and talk about the significance of the individual components of the states. Only certain components of the state are relevant for the present paper, therefore only the properties of these components need be fixed.

Like any other object, states are constructed from elementary objects and selectors and these are now specified.

Elementary objects:

N ... natural numbers and zero
UN ... infinite set of names (referred to as unique names)[1]
TP ... set of textpointers

Selectors:

ID ... set of identifiers
UN ... set of unique names [1]

In addition the following individual selectors are used: s-d, s-dn, s-U, s-tp, s-e, s-eo, s-epa. As mentioned in chapter 2 natural numbers can be considered as selectors for elements of lists.

The states of the twin machine satisfy the predicate is-state as defined below (in the style of /McCarthy 1962/).

(S1) $\text{is-state}(\xi) \supset (\text{is-dump}(\text{s-d}(\xi))\ \&$
$\qquad \text{is-dendir}(\text{s-dn}(\xi))\ \&$
$\qquad \text{s-U}(\xi) \subseteq \text{UN})$

(S2) $\text{is-dump}(d) \supset \text{is-de}(d_i)$ for $1 \leq i \leq l(d)$

(S3) $\text{is-de}(de) \supset (\text{s-tp}(de) \in \text{TP}\ \&$
$\qquad \text{is-env}(\text{s-e}(de))\ \&$
$\qquad \text{is-env}(\text{s-eo}(de))\ \&$
$\qquad \text{s-epa}(de) \in \text{N})$

1) note the double role of unique names.

(S4) $\text{is-env}(e) \supset (D(e) \subseteq ID) \;\&\; (R(e) \subseteq UN)$

(S5) $\text{is-dendir}(dn) \supset (D(dn) \subseteq UN) \;\&\; (R(dn) \subseteq DEN)$

DEN is the set of all possible denotations, only procedure denotations are further specified.

(S6) $\text{is-proc-den}(den) \supset (den \in DEN) \;\&$
$\text{s-tp}(den) \in TP \;\&$
$\text{is-env}(\text{s-e}(den)) \;\&$
$\text{s-epa}(den) \in N)$

Comments on the significance of the state components:

Let ξ be a state (see S1):

s-d(ξ)	list called dump which corresponds to the dynamic sequence of block and procedure activations. The top element top (s-d(ξ)) corresponds to the most recent block or procedure activation.
s-dn(ξ)	denotation directory which associates unique names with their corresponding denotations, i.e. the denotation of a given unique name u is u(s-dn(ξ)).
s-U(ξ)	set of unique names used so far.

Let de be a dump element (see (S3)): such a dump element corresponds to a specific activation of a block or procedure.

s-tp(de)	text pointer to the block or procedure

s-e(de)	environment which contains all referenceable identifiers of the block or procedure and their corresponding unique names for the specific activation. Thus if id is a referenceable identifier then id(s-e(de)) yields its associated unique name.
s-eo(de)	environment which contains all local identifiers of the block or procedure and their corresponding unique names for the activation.
s-epa(de)	natural number which is the index of the dump element corresponding to the environmentally preceding [1] block or procedure.

Let den be a procedure denotation:

s-tp(den)	pointer to the body of the procedure
s-e(den)	environment which resulted from the activation of the block which declared the procedure
s-epa(den)	the index of the dump element corresponding to the block activation in which the procedure was declared (environmentally preceding activation).

All s-e components are used exclusively by the defining model and all s-epa and s-eo components are used exclusively by the implementation model. The other components are common to both models.

[1] Sometimes called statically or lexicographically preceding. The term environment chain is also used below in preference to the alternatives.

The defining model is constructed with the copy rules of ALGOL 60 in mind (i.e. name conflicts of identifiers are resolved by suitable changes). The model deviates slightly from the copy rules in that new names are introduced for all identifiers and not only in cases where conflicts actually occur. Moreover the program text is not modified: instead the correspondence between identifiers and the new names introduced for them is recorded in the so-called environment. All information which may become associated with an identifier during computation (e.g. values of variables, procedures, etc.) is recorded in the denotation directory under the corresponding unique name. The set UN serves as the source for the unique names to be generated and the component U of the state contains all names already used. Procedure denotations are composed of a text pointer to the body of the procedure and the environment which was current in the activation which introduced the procedure. Procedures are passed to parameters by passing their entire denotation.

The implementation model only keeps the local identifiers and their unique names for each activation. The so-called environment chain, established by the s-epa components of each dump element, permits reference to all referenceable identifiers. With this mechanism procedure denotations can be composed of the procedure body and index of the dump which corresponds to the activation which declared the procedure.

3.1.3 Initial states and state transitions

For convenience some additional notational conventions are introduced:

$d \underset{Df}{=} s\text{-}d(\xi)$ thus d_i is the i-th element of the dump of ξ

$dn \underset{Df}{=} s\text{-}dn(\xi)$

$U \underset{Df}{=} s\text{-}U(\xi)$

Components of the top element of the dump of ξ:

$$\begin{aligned}
&tp \ \underset{Df}{=} \ s\text{-}tp(top(d))\\
&e \ \underset{Df}{=} \ s\text{-}e(top(d))\\
&eo \ \underset{Df}{=} \ s\text{-}eo(top(d))\\
&epa \ \underset{Df}{=} \ s\text{-}epa(top(d))
\end{aligned}$$

An arbitrary computation is a sequence of states:

$$\xi^1, \xi^2, .., \xi^n, \xi^{n+1}, \ldots$$

where

ξ^1 ... arbitrary initial state
ξ^n ... state after n steps of the computation

If possible superscripts are avoided by using ξ, ξ' instead of ξ^n, ξ^{n+1}.

Finally the above rules for abbreviating components of ξ may analogously be applied with superscripts, e.g.:

$$e' = s\text{-}e(top(d')) \text{ where } d' = s\text{-}d(\xi')$$

For the initial state of the twin machine it is assumed that the outermost block has been activated and that this block has no declarations 1). This means that the current text pointer, tp, points to the entire program, that there are no referenceable identifiers (empty environment and denotation directories) and that there are no used unique names.

1) This is not a serious restriction because one can always embed programs in such a block.

An <u>intial state</u> ξ^1 has therefore the following properties:

(T1) $l(d^1) = 1$
(T2) $tp^1 = I$
(T3) $e^1 = \Omega$
(T4) $eo^1 = \Omega$
(T5) $epa^1 = 0$
(T6) $dn^1 = \Omega$
(T7) $U^1 = \{\}$

The initial state may also depend on the given input data for the program to be interpreted. However, this fact is irrelevant for the present discussion.

The following postulates on the state transitions are an attempt to characterize only block, procedure activation and the passing of procedures as parameters whilst avoiding restrictions of the considered class of programming languages with respect to other concepts and features they might contain. In particular, only a few assumptions are made about how the machine proceedes to interpret a given program text, e.g. no statement counter is mentioned in the postulates.

For a given program text, tx, the machine starts a computation with a certain initial state. The possible transitions from one state to its successor state are constrained by the following postulates. For mathematical reasons there is no condition under which the machine is said to stop. Instead, when in a given state the attempt is made to terminate the outermost block this state is repeated infinitely many times, so that in the proof only infinite computations have to be considered.

Four cases of state transitions are distinguished:

1. block activation (including the treatment of locally declared identifiers);
2. procedure activation (including the passing of arguments,in particular of procedures);
3. termination of blocks or procedures;
4. other state transitions.

Block and procedure activation have much in common so that it seems worthwhile to factor out these common properties.

The postulates refer to the transition from ξ to ξ'.

<u>Common part of block and procedure activation</u>:

In either case there is a text pointer, tp-b, to the block or procedure body to be activated. In the case of blocks there is no specified way in which tp-b comes into existence because the way in which interpretation proceeds is left in general unspecified. For procedure activations tp-b is part of the denotation of the procedure identifier which caused the call. ID(tx,tp-b) yields the set of local identifiers declared in the block or the set of parameters of the procedure body. An auxiliary environment, eo-b, is constructed from the set of local identifiers by associating new unique names (which are at the same time recorded in the set of used names) with identifiers in the set. This auxiliary environment is then used as the new local environment and for the updating of the environment of the defining machine.

Given: tx, tp-b

Auxiliary environment eo-b:

(T 8) D(eo-b) = ID(tx,tp-b)

(T 9) $R(eo\text{-}b) \cap U = \{\}$

State components:

(T10) rest(d') = d

(T11) tp' = tp-b

(T12) eo' = eo-b

(T13) for $u \in U$ & is-proc-den(u(dn')):

u(dn') = u(dn)

(T14) $U' = R(eo\text{-}b) \cup U$

Comments:

ad (T9): This postulate only guarantees that the unique names used in the auxiliary environment are really new (not in the set of used names). This is sufficient for the proof although to model ALGOL 60 or PL/I correctly one must also guarantee that each identifier is associated with a different new name.

ad (T10): Notice that this also ensures that l(d') = l(d) + 1.

ad (T13): Guarantees that procedure denotations in the new state ξ' having old names have not been modified in any way. The postulate is sufficient in this form for the proof although it permits deletion of existing procedure denotations upon activation. For a complete model of ALGOL 60 or PL/I this would not be the case. No assumptions are made about denotations other than procedure denotations.

For block and procedure activations it is necessary to create a new environment from a given one and the auxiliary environment containing the newly declared local identifiers of the block or procedure. The new environment is supposed to retain all the identifier - unique name pairs unless the identifiers are redeclared. This is achieved by the function "update" defined below.

Definition:

(D1) $\text{update}(e1,e2) = e3$

such that: $$\text{id}(e3) = \begin{cases} \text{id} \in D(e2) \longrightarrow \text{id}(e2) \\ \text{id} \notin D(e2) \longrightarrow \text{id}(e1) \end{cases}$$

for: is-env(e1) and is-env(e2)

The following immediate consequences will be useful:

(C1) $D(\text{update}(e1,e2)) = D(e1) \cup D(e2)$

(C2) $R(\text{update}(e1,e2)) \subseteq R(e1) \cup R(e2)$

The rest of the postulates can now be given.

Block activation:

Given: tp-b, the text pointer to the block to be activated.

(T15) is-block(tp-b(tx))

(T16) $\text{tp-b} \Longleftarrow \text{tp}$

State components:

(T17) $e' = \text{update}(e, \text{eo-b})$

(T18) $\text{epa}' = l(d)$

(T19) for $u \in R(eo\text{-}b)$ & is-proc-den(u(dn')):

a) s-tp(u(dn')) $\Leftarrow$ tp-b
is-body(s-tp(u(dn'))(tx))

b) s-e(u(dn')) = e'

c) s-epa(u(dn')) = l(d')

Comments:

ad (T18):

For blocks the dynamically preceding activation is also the environmentally preceding one.

ad (T19): This postulate takes care of the procedure denotations that may have been introduced by local declarations within the block.

Procedure activation:

Given: id-p, the identifier of the procedure to be activated.

(T20) id-p $\in$ D(e)

(T21) is-proc-den(u-p(dn))

Abbreviations:

(T22) u-p = id-p(e)
tp-b = s-tp(u-p(dn))
e-p = s-e(u-p(dn))
epa-p = s-epa(u-p(dn))

State components:

(T23) e' = update(e-p,eo-b)

(T24) epa' = epa-p

(T25) for $u \in R(eo\text{-}b)$ & is-proc-den(u(dn')):
$\exists u1(u1 \in R(e)$ & $u(dn') = u1(dn))$

Comments:

ad (T24): The environmentally preceding activation is the one in which the procedure was declared and its denotation introduced.

ad (T25): Parameters are treated as the only local identifiers introduced by procedure activations. The postulate is concerned with passing procedures as arguments to parameters. No further assumptions are made on argument passing.

<u>Termination of block and procedure activations</u>:

<u>$l(d) > 1$</u>:

(T26) $d' = rest(d)$

(T27) for $u \in U$ & is-proc-den(u(dn')):
$u(dn') = u(dn)$

(T28) $U' = U$

<u>$l(d) = 1$</u>:

(T29) $\xi' = \xi$

Comments:

ad (T29): This case can be interpreted as an attempt to close the outermost block, i.e. the end of program execution. In this case, the state is repeated indefinitely in order to make all computations infinite.

Other state transitions:

(T30) $d' = d$

(T31) for $u \in U$ & is-proc-den(u(dn')):
$u(dn') = u(dn)$

(T32) $U' = U$

3.2 Formulation of the Equivalence Problem

The equivalence problem in question is concerned with the reference to identifiers, more precisely with the interpretation of the use of identifiers in the given program text.

It is assumed that reference to an identifier only ever involves reference to or change of the corresponding denotation in the denotation directory. Thus the environment in the defining model and the local environment and epa component of the implementation model are solely auxiliary devices to make the reference to the denotation directory possible. The unique names are irrelevant except for this purpose.

Therefore, the two reference mechanisms are considered to be equivalent if for any given state they yield the same unique name for any identifier, thus relating any identifier to the same entry in the denotation directory.

Reference mechanism of the defining model:

The unique name which corresponds to a given referenceable identifier id for some state ξ is simply:

$$id(e) \qquad \text{for } id \in D(e)$$

Reference mechanism 1 for the implementation model:

The simplest way to get the unique name for a given referenceable identifier, using the implementation model, is to search along the environment chain to find the activation which introduced the identifier and its unique name. Two auxiliary functions [1] are introduced to accomplish this: the function s(id,d) which yields the number of steps along the environment chain to the activation which introduced the identifier; the function index(n,d) which yields the index of the dump element n steps back along the environment chain.

(D2) $s(id,d) = (id \in D(eo) \longrightarrow 0, s(id,rest(d,epa)) + 1)$

(D3) $index(n,d) = (n = 0 \longrightarrow l(d), index(n-1,rest(d,epa)))$

where: $epa = s\text{-}epa(top(d))$
$eo = s\text{-}eo(top(d))$

The actual reference mechanism is:

$id(s\text{-}eo(d_{index(s(id,d),d)}))$ for $id \in D(e)$

Thus the first equivalence problem is:

for $id \in D(e)$:
$id(e) = id(s\text{-}eo(d_{index(s(id,d),d)}))$

This result is proved as Theorem I in section 3.4.

Assumption:

It is assumed that occurences of identifiers are only interpreted if the block or body which immediately contains them is the current one (see section 3.1.1).

1) Recursive functions which rely on the environment chain are justified because Lemma 7 holds.

Under this assumption it turns out, as expected, that s(id,d) depends only on the given program text, i.e. the number of steps which will need to be made along the environment chain can be statically determined for each use of an identifier (see Theorem III).

Reference mechanism 2 of the implementation model:

This mechanism differs from the previous one in the method by which the index of the required element is computed.

A list called disp (for display) is introduced which contains indices of the dump elements of the environment chain. The display is used to determine the actual index for a given identifier.

Definitions:

(D4) $le(d) = (epa = 0 \longrightarrow 0,\ le(rest(d,epa)) + 1)$
where: $epa = s\text{-}epa(top(d))$

(D5) $depth(id,d) = le(rest(d,index(s(id,d),d)))$

(D6) $disp_i = index(le(d)-i,d) \qquad \text{for } 0 \leq i \leq le(d) - 1$

The second reference mechanism is:

$$id(s\text{-}eo(d_{disp_{depth(id,d)}})) \qquad \text{for } id \in D(e)$$

The second equivalence problem is thus:
for $id \in D(e)$:

$$id(e) = id(s\text{-}eo(d_{disp_{depth(id,d)}}))$$

This result is proved as Theorem II in section 3.4.

The function depth is, under the above assumption, only a function of the program text (see Theorem III) as explained in /Dijkstra 1960/ who also introduced the display.

The display is usually kept as a state component and must be updated. It is sufficient but not always necessary to update the display every time the dump changes. The subject of optimization is discussed in /Henhapl, Jones 1970/.

The question may arise as to how relevant the presented implementation model is to actual implementations. Implementations as usually described (e.g. /Randell, Russell 1964/) do not use unique names and denotation directories but instead keep the information directly in the dump (stack). They use relative positions within each dump element instead of identifiers for access. Furthermore, procedure denotations, except for parameters, have not to be kept in the dump at all since the text pointer can be computed statically and the index of the environmentally preceding activation can be computed in the same way as is already done for all other identifiers.

These deviations can be overcome by the following simple considerations. It is assumed that each identifier when first introduced is associated with a unique name different from the unique names chosen for all other identifiers. It follows that a unique name occurs at most once in the local environments of the dump. Therefore the indirect step via the unique names is unnecessary. Since the only interesting entries of the denotation directory are those referred to by names occuring in some local environment of the dump, one can omit the denotation directory altogether and associate denotations directly with identifiers in the local environment. The introduction of relative positions instead of identifiers amounts to a trivial local change of names. The usual implementation where references are made via the integer pair (depth of nesting, relative position) is therefore derivable from the present model.

Apart from the final models being close to actual implementations, the lemmas and theorems give a clear insight into the structure of the state components and their relations.

3.3 Proof Principles

The basic proof principle in the subsequent proofs is induction on the number of steps of computations, i.e. a proof $P(\xi^n)$ for all n could in general take the form:

basis: $P(\xi^1)$

induction step: a) $P(\xi^n) \supset P(\xi^{n+1})$

or b) $\underset{m<n}{\forall} P(\xi^m) \supset P(\xi^n)$

To prove the induction step it is in general necessary to make four case distinctions according to the distinguished types of state transitions.

If the property to be proved is a property of the dump component alone, $P(d^n)$, it is valid to use induction on the length of the dump. To show this, the following two auxiliary lemmas are useful:

(L1) for $1 \leq i < l(d^n)$:

$\exists m(d^m = rest(d^n,i) \;\&\; 1 \leq m < n)$

(L2) $rest(d^n,1) = d^1$

Both lemmas can be proved by induction on the steps of computations.[1)]

[1)] It is felt that that inclusion of the detailed proofs at this point would detract from the development of the section. The interested reader may find them in Appendix I.

One can think of the set of all possible dumps as being generated from the dumps of the initial states and the state transitions corresponding to block and procedure activation only. This is valid because Lemma 1 shows that termination, for $l(d) > 1$, does not add new elements to the generated set and from (T29) and (T30) it follows that the same is true for all other cases. From (T10) it follows that the rest of a dump generated by block and procedure activation is always identical to the dump of the preceding state. Therefore the following induction principle is valid for proving say P(d) for the specific machine and the dump component of its states:

basis: $P(d^1)$

induction step: a) $P(rest(d)) \supset P(d)$

or

b) $\underset{i<l(d)}{\forall} P(rest(d,i)) \supset P(d)$

where for the induction step only the cases of block and procedure activations have to be considered.

Clearly the proof method also holds if P(d) can be justified without appeal to the induction hypotheses. Furthermore, Lemma 1 states that for each instance of i there exists a predecessor state which is identical to rest(d,i). Therefore from a given assertion, P(d), it is valid to conclude:

$$\underset{i<l(d)}{\forall} P(rest(d,i))$$

3.4 Proofs [1]

Section 3.2 has introduced the two equivalence properties in which we are interested. These results, Theorems I and II, Theorem III which shows how certain indexes can be statically computed and a number of lemmas used in the argument, are proved in this section.

Lemmas 3 and 4 show that:
the set of used unique names does not lose elements; procedure denotations whose names are in U cannot be changed to different procedure denotations.

These results are required in Lemmas 5 and 6 where the inductive step based on procedure activation may rely on properties of any preceding element of the computation.

Lemma 3:

(L3) for $1 \leq m < n$:

$$U^{m} \subseteq U^{n}$$

Proof:

Follows immediately from (T14),(T28),(T29),(T32)

Lemma 4:

(L4) for $1 \leq m < n \;\&\; u \in U^{m}$:

$$\text{is-proc-den}(u(dn^{n})) \supset u(dn^{n}) = u(dn^{m})$$

Proof:

Follows immediately from (L3),(T13),(T27),(T29),(T31)

---

[1] Case distinctions are identified by underlining and indentation. Appendix II contains a list of all formulae to which reference is made.

Lemma 5 shows that the range of the current environment is contained in U. The proof of the lemma [1] is by induction on steps of the computation using Lemmas 1, 3 and 4 as well as the relevant state transition postulates. Part b) of the statement is included to make the induction for the case of procedure activation possible. The result is required in the proof of Lemma 6 to connect the fact that an identifier is in the domain of some environment component with the hypotheses of Lemma 4.

(L5) a) $R(e) \subseteq U$

b) for $u \in U$ & is-proc-den(u(dn)):
$R(s\text{-}e(u(dn))) \subseteq U$

Lemma 6 states that any procdure denotation referencable from some environment has an epa component not greater than the index of the dump containing that environment, and that the other components are those corresponding to the dump element which generated the denotation. The lemma is proved [1] by induction on the steps of the computation using Lemmas 1,4 and 5 as well as the relevant state transition postulates. The result is required to establish the argument for the case of procedure activation in Lemmas 7, 8 and 10.

(L6) for $u \in R(e)$ & is-proc-den(u(dn)):

a) $1 \leq s\text{-}epa(u(dn)) \leq l(d)$

b) $s\text{-}e(u(dn)) = s\text{-}e(d_{s\text{-}epa(u(dn))})$

c) $s\text{-}tp(u(dn)) \Leftarrow s\text{-}tp(d_{s\text{-}epa(u(dn))})$

[1] The proofs of Lemmas 5 and 6 have been put in Appendix I because the reader who wishes to limit his reading of detailed proofs might find the subsequent ones more useful.

Lemma 7 establishes that epa components point to an earlier element of the dump than that where they are contained. (The proof uses the result of section 3.3 that it is only necessary to consider block and procedure activations). The lemma is required in order to show, in Theorem I, and Lemma 9 that the preceding element of the environment chain is in the range of the induction hypotheses.

<u>Lemma 7</u>:

(L7) for d such that $l(d) > 1$:

$1 \leq epa < l(d)$

<u>Proof</u> for top (d')

1	<u>top(d') generated by block activation</u>:	
2	$epa' = l(d)$	(T18)
3	$1 \leq epa' < l(d')$	2,(T10)
4	<u>top(d') generated by procedure activation</u>:	
5	$u\text{-}p \in R(e)$	(T20),(T22)
6	is-proc-den(u-p(dn))	(T21)
7	$1 \leq s\text{-}epa(u\text{-}p(dn)) \leq l(d)$	5,6,(L6a)
8	$1 \leq epa' < l(d')$	(T24),(T22),7,(T10)

Lemma 8 expresses the fundamental connection between the environment and search mechanisms: it shows that the environment component of an element of the dump differs from the environment component of the environmentally preceding dump element by exactly the local environment of the current element. (The proof uses the result of section 3.3 that it is only necessary to consider block and procedure activations). The lemma makes it possible to prove Theorem I without the case distinction of the generating state transitions.

Lemma 8:

(L8) for d such that $l(d) > 1$:

$$e = \text{update}(s\text{-}e(d_{epa}), eo)$$

Proof for top(d')

1	top(d') generated by block activation:	
2	$e' = \text{update}(e, eo\text{-}b)$	(T17)
3	$= \text{update}(e, eo')$	2,(T12)
4	$= \text{update}(s\text{-}e(d'_{l(d)}), eo')$	3,(T10),(N1)
5	$= \text{update}(s\text{-}e(d'_{epa'}), eo')$	4,(T18)
6	top(d') generated by procedure activation:	
7	$1 \le epa\text{-}p \le l(d)$	(T21),(T22),(T20),(L6a)
8	$e\text{-}p = s\text{-}e(d_{epa\text{-}p})$	(T21),(T22),(T20),(L6b)
9	$= s\text{-}e(d'_{epa\text{-}p})$	7,8,(T10),(N1)
10	$e' = \text{update}(e\text{-}p, eo\text{-}b)$	(T23)
11	$= \text{update}(e\text{-}p, eo')$	10,(T12)
12	$= \text{update}(s\text{-}e(d'_{epa\text{-}p}), eo')$	9,11
13	$= \text{update}(s\text{-}e(d'_{epa'}), eo')$	12,(T24)

Theorem I justifies the first result presented in section 3.2. (Its proof uses the induction on dumps discussed in section 3.3). The result is further elaborated in Theorem III, where it is shown that an index can be computed from the text corresponding to the result of "s", and is also used as a basis of Theorem II.

<u>Theorem I</u>:

for $id \in D(e)$:

$$id(e) = id(s\text{-}eo(d_{index(s(id,d),d)}))$$

<u>Proof</u> by induction on the dump

<u>basis</u>:

1	$s\text{-}e(rest(d,1)) = e^1$	(L2)
2	vacuously true	1,(T3)

<u>induction step</u>:

3	$l(d) > 1$	
4	$id \in D(e)$	Hyp
5	$D(e) = D(s\text{-}e(d_{epa})) \cup D(eo)$	3,(L8),(C1)
6	<u>$id \in D(eo)$</u>:	
7	$id(e) = id(eo)$	3,(L8),6,(D1)
8	$s(id,d) = 0$	6,(D2)
9	$index(s(id,d),d) = l(d)$	8,(D3)
10	$id(s\text{-}eo(d_{index(s(id,d),d)})) = id(eo)$	9
11	$id(e) = id(s\text{-}eo(d_{index(s(id,d),d)}))$	7,10

12	$\underline{id \notin D(eo):}$	
13	$id(e) = id(s\text{-}e(d_{epa}))$	3,(L8),12,(D1)
14	$s(id,d) = s(id,rest(d,epa)) + 1$	12,(D2)
15	$index(s(id,d),d) = index(s(id,rest(d,epa)),rest(d,epa))$	14,(D3)
16	$1 \leq epa < l(d)$	3,(L7)
17	$id \in D(s\text{-}e(d_{epa}))$	12,5,4
18	$id(s\text{-}e(d_{epa})) = id(s\text{-}eo(d_{index(s(id,rest(d,epa)),rest(d,epa))}))$	16,17,IH
19	$id(e) = id(s\text{-}eo(d_{index(s(id,d),d)}))$	4,18,13,15

Lemma 9 establishes a relation which holds between the index and le functions, since both follow exactly the same chain. The result is used in Theorem II to provide the link to the result of Theorem I.

Lemma 9:

(L9) for $0 \le i \le le(d)$:

$$le(rest(d,index(i,d))) = le(d) - i$$

Proof by induction on l(d):

basis:

1	$l(d) = 1$	
2	$epa = 0$	1,(L2),(T5)
3	$le(d) = 0$	2,(D4)
4	$i = 0$	3
5	$rest(d,index(0,d)) = d$	(D3)
6	$le(rest(d,index(i,d))) = le(d) - i$	4,5

induction step:

7	$l(d) > 1$	
8	$0 \le i \le le(d)$	Hyp
9	$epa \ge 1$	7,(L7)
10	$i > 0$:	
11	$le(d) = le(rest(d,epa)) + 1$	9,(D4)
12	$0 \le i - 1 \le le(rest(d,epa))$	8,10,11
13	$le(rest(rest(d,epa),index(i-1,rest(d,epa)))) = le(rest(d,epa)) - (i-1)$	12,IH
14	$le(rest(d,index(i,d))) = le(d) - i$	13,(D3),(D4)
15	$i = 0$:	
16	$le(rest(d,index(i,d))) = le(d) - i$	15,4,6

Theorem II justifies the second result presented in section 3.2. The result is further elaborated in Theorem III where it is shown that depth can be computed from the text. Section 3.2 has pointed out that disp can be modelled by a state component.

Theorem II:

for $id \in D(e)$:

$$id(e) = id(s\text{-}eo(d_{disp_{depth(id,d)}}))$$

Proof:

1 $disp_{depth(id,d)} =$ (D6),(D5)

$\quad index(le(d) - le(rest(d,index(s(id,d),d))),d)$

2 $\quad = index(s(id,d),d)$ 1,(L9)

3 $id \in D(e)$: Hyp

4 $id(e) = id(s\text{-}eo(d_{disp_{depth(id,d)}}))$ 3,Theorem I,2

Lemma 10 shows that the tp component of a dump element is the successor of the tp component of the environmentally preceding dump element. (The proof uses the result of section 3.3 that it is only necessary to consider block and procedure activations). This lemma, extended over entire environment chains, is used in Theorem III.

Lemma 10:

(L10) for d such that $l(d) > 1$:

$$tp \Longleftarrow s\text{-}tp(d_{epa})$$

Proof for top(d')

top(d') generated by block activation:

1	$tp\text{-}b \Leftarrow tp$	(T16)
2	$\Leftarrow s\text{-}tp(d'_{l(d)})$	1,(T10),(N1)
3	$\Leftarrow s\text{-}tp(d'_{epa'})$	2,(T18)
4	$tp' \Leftarrow s\text{-}tp(d'_{epa'})$	3,(T11)

top(d') generated by procedure activation:

5	$epa\text{-}p \leq l(d)$	(T20),(T22),(T21),(L6a)
6	$tp\text{-}b \Leftarrow s\text{-}tp(d_{epa\text{-}p})$	(T20),(T22),(T21),(L6c)
7	$\Leftarrow s\text{-}tp(d'_{epa\text{-}p})$	6,5,(T10),(N1)
8	$\Leftarrow s\text{-}tp(d'_{epa'})$	7,(T24)
9	$tp' \Leftarrow s\text{-}tp(d'_{epa'})$	8,(T11)

Lemma 11 establishes that the relationship between the identifiers in ID(tx,tp) and in the component eo is preserved. (The proof uses the result of section 3.3 that it is only necessary to consider block and procedure activations). The result is used in Theorem III.

Lemma 11:

(L11) D(eo) = ID(tx,tp)

Proof:

1 basis: l(d) = 0

2 D(e) = {} 1,(L2),(T3)

3 ID(tx,I) = {} Outermost block 3.1.1

4 D(e) = ID(tx,tp) 1,(L2),(T2),2,3

induction: for top (d')

5 D(eo') = D(eo-b) (T12)

6 = ID(tx,tp-b) 5,(T8)

7 = ID(tx,tp') 6,(T11)

The functions"s" and "depth" are shown above as functions of the dump. However, this was done only to facilitate the above proofs and Theorem III shows that the, anticipated, transition to functions of the text is possible. Values computed by a compiler could then be associated with the references. Since the result relates to the program texts, which have not been formally characterized, the argument is less formal than preceding proofs.

Theorem III:

Equivalent functions to "s" and "depth" can be found which depend only on the static text.

Justification:[1]

Using the generalization principle given in section 3.3:

1 $l(d) > index(1,d) > index(2,d) > \dots > index(n,d) = 1$ (L7),(D3)

2 $tp \Leftarrow s\text{-}tp(d_{index(1,d)}) \Leftarrow s\text{-}tp(d_{index(2,d)}) \Leftarrow \dots \Leftarrow I$ 1,(L10),(T2)

let $tp_1, tp_2, \dots, I$ be the pointers to the blocks in which $tp(tx)$ is nested, section 3.1.1 shows that:

3 $tp \Leftarrow tp_1 \Leftarrow \dots \Leftarrow I$

and also, since such chains are unique:

4 $tp_i = s\text{-}tp(d_{index(i,d)})$ 2,3

5 $id \in D(s\text{-}eo(d_{index(i,d)})) \equiv id \in ID(tx, s\text{-}tp(d_{index(i,d)}))$ (L11)

6 $\equiv id \in ID(tx, tp_i)$ 5,4

1) The function index is used as a notational convenience in the following argument.

7 $s(id,d) = (id \in D(eo) \longrightarrow 0, s(id,rest(d,\dot{epa})))$ (D2)

Given an occurence (id,tp-id) let tp-o be the pointer to the block or procedure body immediately containing it (see section 3.1.1). Section 3.2 (assumption) shows that occurences are only referenced when the block or body which immediately contains them is the current one, then:

8 $tp\text{-}o = tp$

let:

9 $s'(id,tp) = (id \in ID(tx,tp) \longrightarrow 0, s'(id,tp'))$
where $tp \Longleftarrow tp'$

It can be shown by induction:

10 $s(id,d) = s'(id,tp\text{-}o)$ 7,8,6,9

11 "s'" is the equivalent function to "s" which depends only on the text.

A similar argument results in discovery of the required static function for depth.

3.5 Discussion

A number of aspects of the above example can now be discussed. The postulates on the state transitions, which form the basis of the above proof, are in some sense too restrictive. The condition that any "other statement" does not alter the stack or procedure denotations has been sufficient for constructing the proof but is clearly not necessary to achieve the same result. This is not a serious constraint and where it need be violated (e.g. a sufficiently restricted type of procedure variable) the proof can be extended in a straightforward manner.

It is perhaps worth pointing out how the above example differs from the previous papers on blocks (see section 4). The definition of the state transitions by postulates avoids the argument from the control in /Lucas 1968/ and the necessity to give too much detail in the explicit state transition functions of /Henhapl, Jones 1970/. The result proved above goes further than that in /Lucas 1968/ and is made simpler than in /Henhapl, Jones 1970/ by the idea of using the function "s", to compute the number of steps for "index", and showing, in a subsequent step, that it can be computed statically.

The proofs have always used a more or less explicit twin machine. However, the use of the derived proof principles (see section 3.3) appears to be a worthwhile extension and to be the kind of result which should be sought in future work.

The experience gained with this example suggests that, if the correctness proof is to be based on the equivalence of two abstract machines, there are certain essential steps in the argumentation. Although shorter proofs may be found, the authors foresee no conceptually simpler proof unless an appropriate, completely thought out, basis for the language definitions can be found.

4. OTHER RESULTS ON IMPLEMENTATIONS AND THEIR JUSTIFICATION

The interest in proofs of correctness of implementation methods was started at the Vienna Laboratory with /Lucas 1968/ and work has continued on the subject of blocks. However, implementation methods have been documented and justified for other aspects of languages and this section reviews the main results.

It will set the context for the following discussion if it is mentioned that the emphasis, so far, has been to consider implementation problems as seen from their object time organisation and to postpone consideration of the compile time processing.

4.1 Blocks

In the initial attempt to show the equivalence of two reference mechanisms, /Lucas 1968/, the base reference method is the environment mechanism used in the current paper. The alternative method is a search for the first element of the static chain where the identifier occurs in the local environment component (i.e. a function similar to "s" above, but which returns the index of the dump). The problem considered is to prove equivalent the two mechanisms from definitions in the "Vienna Method" notation. Thus the relations on which the proof is based are derived from the model. The proof itself, which introduced the twin machine concept used above, is divided in much the same way as Lemma 8 and Theorem 1 of the current paper.

Most of the remaining work on reference mechanisms is presented in /Henhapl and Jones 1970/. In that report the possibility of block termination caused by goto statements is also considered. The base mechanism is the copy rule (see /Naur et al. 1962/) and display models, both of the type described above and of the type using unique static block names, are proved correct. A number of optimizations to the basic methods are also discussed.

Intermediate models are included so that from a number of lemmas about one model a theorem is proved which establishes the equivalence to the next model in a chain of equivalences. Some problems resulting from the choice of intermediate models are discussed in the summary of /Henhapl and Jones 1970/. A further comment on the steps made is that the direct proof of the equivalence of the display to the search models (using axioms of the text) appears less elegant than the approach adopted above. The definition of the state transitions by functions facilitates the derivation of relations required for the proofs, but still gives much unnecessary detail.

4.2 Loops

The problem of correctness proofs for optimized implementations of loop handling is discussed in /Zimmermann 1970/. This report begins by illustrating how most interpretations for loops can be represented by state transitions which:

initialize (control variable etc.)
execute body of the loop
test (for repeat condition)
update (control variable etc.)
exit

Given two sets of such transition functions a specific way of proving their equivalence is shown. This technique requires that relations are found which fulfil the role of Induction Hypotheses, but the proof itself consists only of proving certain lemmas which do not require induction. The technique has been used to show that assignment statements can be moved out of loops (providing certain constraints are satisfied) and that the evaluation of expressions, linear with respect to the control variable, can be handled in an efficient way.

The technique of factoring out the inductive reasoning for whole sets of proofs should be capable of generalization.

4.3 Storage

The properties of storage are defined in /Walk et al. 1969/ by axioms. Thus the correctness of an implementation is established by showing that it satisfies the axioms, furthermore finding any such model proves that the axioms are consistent. These questions are considered in /Henhapl 1969/ but more work is required both on the axioms and relevant models.(The paper in the current volume /Bekič and Walk 1970/ is a contribution to the former area).

4.4 Expressions

Expression evaluation techniques are described extensively in the literature, a study of which has led to an extensive bibliography /Chroust 1970/. Work is now in progress on describing their object time operations in a uniform formal style. The value of progressing to correctness proofs is questionable since the compile time aspects of many of the techniques would be a more likely source of error in actual implementations.

4.5 Bases for Proof Construction

A number of experiments have been made as to the possibility of basing the proofs on some other style of definition. However, apart from the minor changes made during the sequence of block papers, no completely thought out proposal has proved acceptable.

APPENDIX I:

This appendix contains the detailed **proofs** of Lemmas 1,2,5 and 6.

Lemma 1:

(L1) for $1 \leq i < l(d^n)$:
$\exists m(d^m = rest(d^n,i) \ \& \ 1 \leq m < n)$

Proof by induction on steps of the computation.

basis: ξ^1

1 lemma is vacuously true (T1)

induction step: $\xi^n \longrightarrow \xi^{n+1}$

block or procedure activation:

2	$1 \leq i < l(d^{n+1})$:	Hyp
3	$\underline{1 \leq i < l(d^n)}$:	
4	$\exists m(d^m = rest(d^n,i) \ \& \ 1 \leq m < n)$	3,IH
5	$\exists m(d^m = rest(d^{n+1},i) \ \& \ 1 \leq m < n+1)$	4,(T10),(N1)
6	$\underline{i = l(d^n)}$:	
7	$d^n = rest(d^{n+1},i)$	6,(T10)
8	$\exists m(d^m = rest(d^{n+1},i) \ \& \ 1 \leq m < n+1)$	2,5,7

termination:

case distinction according to state transition

9 $\underline{l(d^{n}) > 1}$:

10 $1 \leq i < l(d^{n+1})$ Hyp

11 $\exists\, m(d^{m} = rest(d^{n+1}, i)\ \&\ 1 \leq m < n + 1)$ 10,IH,(T26),(N1)

12 $\underline{l(d^{n}) = 1}$:

13 immediate from (T29) 12

other state transitions:

14 immediate from (T30)

Lemma 2:

(L2) $\text{rest}(d^n,1) = d^1$

Proof by induction on steps of the computation.

basis: ξ^1

1 lemma is identically true (T1)

induction step: $\xi^n \longrightarrow \xi^{n+1}$

termination:

case distinction according to state transition

2 $l(d^n) > 1$:

3 $\text{rest}(d^{n+1},1) = d^1$ 2,IH,(T26),(N1)

4 $l(d^n) = 1$:

5 immediate from (T29)

all other state transitions (including activation):

6 $\text{rest}(d^{n+1},1) = \text{rest}(d^n,1)$ (T10),(T30)

7 $\text{rest}(d^{n+1},1) = d^1$ IH,6

procedure activation:

15	a) $u\text{-}p \in U$	(T20),(T22),IHa
16	$R(e\text{-}p) \subseteq U$	15,(T21),IHb,(T22)
17	$R(e') \subseteq R(e\text{-}p) \cup R(eo\text{-}b)$	(T23),(C2)
18	$\subseteq U \cup R(eo\text{-}b)$	17,16
19	$\subseteq U'$	18,(T14)
20	b) $u \in U'$ & is-proc-den(u(dn'))	Hyp
	case distinction according to partitioning of U'	(T14)
21	$u \in R(eo\text{-}b)$:	
22	$\exists u1(u1 \in R(e)$ & $u(dn') = u1(dn))$	20,21,(T25)
	let u1 be such a name	
23	$u1 \in U$	22,IHa
24	$R(s\text{-}e(u1(dn))) \subseteq U$	23,20,22,IHb
25	$R(s\text{-}e(u(dn'))) \subseteq U$	24,22
26	$\subseteq U'$	25,(L3)
27	$u \in U$:	
28	is-proc-den(u(dn))	27,20,(L4)
29	$R(s\text{-}e(u(dn))) \subseteq U$	27,28,IHb
30	$R(s\text{-}e(u(dn'))) \subseteq U'$	29,27,20,(L4),(L3)

<u>termination</u>:

31	a) $\exists m(d^m = rest(d)\ \&\ 1 \le m < n)$ let m be such an index	(L1)
32	$d' = d^m$	31,(T26)
33	$R(e') \subseteq U^m$	32,IHa
34	$R(e') \subseteq U'$	31,33,(L3)
35	b) $u \in U'\ \&\ is\text{-}proc\text{-}den(u(dn'))$	Hyp
36	$u \in U$	35,(T28)
37	$u(dn') = u(dn)$	35,36,(T27)
38	$R(s\text{-}e(u(dn))) \subseteq U$	36,35,37,IHb
39	$R(s\text{-}e(u(dn'))) \subseteq U'$	38,37,(L3)

<u>other state transitions</u>:

a) and b) immediate from (T30),(T31),(T32)

Lemma 6:

(L6) for $u \in R(e)$ & is-proc-den(u(dn)):

a) $1 \leq$ s-epa(u(dn)) $\leq$ l(d)

b) s-e(u(dn)) = s-e($d_{\text{s-epa(u(dn))}}$)

c) s-tp(u(dn)) $\Leftarrow$ s-tp($d_{\text{s-epa(u(dn))}}$)

Proof by induction on the steps of the computation

basis: ξ^1

1	lemma is vacuously true	(T3)

induction step: $\xi^1, \xi^2, \ldots, \xi \longrightarrow \xi'$

2	$u \in R(e')$ & is-proc-den(u(dn'))	Hyp
	block activation:	
3	$R(e') \subseteq R(e) \cup R(eo\text{-}b)$	(T17),(C2)
	case distinction according to partitioning of R(e')	3
4	$u \in R(eo\text{-}b)$:	
5	s-epa(u(dn')) = l(d')	4,2,(T19)
6	$1 \leq$ s-epa(u(dn')) $\leq$ l(d')	5
7	s-e(u(dn')) = s-e($d'_{\text{s-epa(u(dn'))}}$)	4,2,(T19),5
8	s-tp(u(dn')) $\Leftarrow$ s-tp($d'_{\text{s-epa(u(dn'))}}$)	4,2,(T19),(T11),5
9	$u \in R(e)$:	
10	$u \in U$	9,(L5a)
11	u(dn') = u(dn)	10,2,(L4)
12	s-epa(u(dn)) $\leq$ l(d)	9,2,11,IHa

13	$1 \le \text{s-epa}(u(dn')) \le l(d')$	11,12
14	$\text{s-e}(u(dn')) = \text{s-e}(d'_{\text{s-epa}(u(dn'))})$	IHb,12,(N1)
15	$\text{s-tp}(u(dn')) \Leftarrow \text{s-tp}(d'_{\text{s-epa}(u(dn'))})$	IHc,12,(N1)
	<u>procedure activation</u>:	
16	$R(e') \subseteq R(\text{e-p}) \cup R(\text{eo-b})$	(T23),(C2)
	case distinction according to partitioning of $R(e')$	16
17	<u>$u \in R(\text{eo-b})$</u>:	
18	$\exists\, u1(u1 \in R(e)\ \&\ u(dn') = u1(dn))$	17,2,(T25)
	let u1 be such a name	
19	$\text{s-epa}(u1(dn)) \le l(d)$	18,2,IHa
20	$1 \le \text{s-epa}(u(dn')) \le l(d')$	18,19
21	$\text{s-e}(u(dn')) = \text{s-e}(d'_{\text{s-epa}(u(dn'))})$	IHb,19,(N1)
22	$\text{s-tp}(u(dn')) \Leftarrow \text{s-tp}(d'_{\text{s-epa}(u(dn'))})$	IHb,19,(N1)
23	<u>$u \in R(\text{e-p})$</u>:	
24	$\text{u-p} \in R(e)\ \&\ \text{is-proc-den}(\text{u-p}(dn))$	(T20),(T22),(T21)
25	$\text{epa-p} \le l(d)$	24,IHa,(T22)
26	$\exists\, m(d^m = \text{rest}(d,\text{epa-p})\ \&\ 1 \le m < n)$	25,(L1)
	let m be such an index	
27	$\text{e-p} = e^m$	26,24,IHb,(T22)
28	$u \in U^m$	27,(L5a)
29	$u(dn^m) = u(dn')$	28,2,(L4)
30	$\text{s-epa}(u(dn^m)) \le l(d^m)$	23,27,2,29,IHa
31	$1 \le \text{s-epa}(u(dn')) \le l(d')$	29,30,26,25
32	$\text{s-e}(u(dn')) = \text{s-e}(d'_{\text{s-epa}(u(dn'))})$	IHb,30,(N1)
33	$\text{s-tp}(u(dn')) \Leftarrow \text{s-tp}(d'_{\text{s-epa}(u(dn'))})$	IHc,30,(N1)

termination:

	case distinction according to state transitions	
34	$\underline{l(d) > 1}$:	
35	$\exists\, m(d^m = rest(d)\ \&\ 1 \le m < n)$	(L1)
	let m be such an index	
36	$u \in R(e^m)$	2,35
37	$u \in U^m$	36,(L5a)
38	$u(dn') = u(dn^m)$	37,2,(L4)
39	$s\text{-}epa(u(dn^m)) \le l(d^m)$	36,2,38,IHa
40	$1 \le s\text{-}epa(u(dn')) \le l(d')$	39,38,35
41	$s\text{-}e(u(dn')) = s\text{-}e(d'_{s\text{-}epa(u(dn'))})$	IHb,39,(N1)
42	$s\text{-}tp(u(dn')) \in s\text{-}tp(d'_{s\text{-}epa(u(dn'))})$	IHc,39,(N1)
43	$\underline{l(d) = 1}$:	
44	immediate from	(T29)

other state transitions:

45	immediate from	(T30),(T31),(T32)

APPENDIX II:

This appendix contains all of the formulae referenced in the paper.

(T 1) $l(d^1) = 1$

(T 2) $tp^1 = I$

(T 3) $e^1 = \Omega$

(T 4) $eo^1 = \Omega$

(T 5) $epa^1 = 0$

(T 6) $dn^1 = \Omega$

(T 7) $U^1 = \{\}$

(T 8) $D(eo\text{-}b) = ID(tx,tp\text{-}b)$

(T 9) $R(eo\text{-}b) \cap U = \{\}$

(T10) $rest(d') = d$

(T11) $tp' = tp\text{-}b$

(T12) $eo' = eo\text{-}b$

(T13) for $u \in U$ & is-proc-den(u(dn')):
    $u(dn') = u(dn)$

(T14) $U' = R(eo\text{-}b) \cup U$

(T15) is-block(tp-b(tx))

(T16) $tp\text{-}b \Longleftarrow tp$

(T17) $e' = update(e,eo\text{-}b)$

(T18) $epa' = l(d)$

(T19) for $u \in R(eo\text{-}b)$ & is-proc-den(u(dn')):

a) $s\text{-}tp(u(dn')) \Longleftarrow tp\text{-}b$
   is-body(s-tp(u(dn'))(tx))

b) $s\text{-}e(u(dn')) = e'$

c) $s\text{-}epa(u(dn')) = l(d')$

(T20) $id\text{-}p \in D(e)$

(T21) is-proc-den(u-p(dn))

(T22) u-p = id-p(e)
tp-b = s-tp(u-p(dn))
e-p = s-e(u-p(dn))
epa-p = s-epa(u-p(dn))

(T23) e' = update(e-p,eo-b)

(T24) epa' = epa-p

(T25) for $u \in R(eo\text{-}b)$ & is-proc-den(u(dn')):

$\exists u1(u1 \in R(e)$ & $u(dn') = u1(dn))$

(T26) d' = rest(d)

(T27) for $u \in U$ & is-proc-den(u(dn')):
u(dn') = u(dn)

(T28) U' = U

(T29) $\xi' = \xi$

(T30) d' = d

(T31) for $u \in U$ & is-proc-den(u(dn')):
u(dn') = u(dn)

(T32) U' = U

(D 1) update(e1,e2) = e3

such that: $$id(e3) = \begin{cases} id \in D(e2) \longrightarrow id(e2) \\ id \notin D(e2) \longrightarrow id(e1) \end{cases}$$
for: is-env(e1) and is-env(e2)

(D 2) $s(id,d) = (id \in D(eo) \longrightarrow 0,\ s(id,rest(d,epa)) + 1)$

(D 3) $index(n,d) = (n = 0 \longrightarrow 1(d),\ index(n-1,rest(d,epa)))$
where: epa = s-epa(top(d))
eo = s-eo(top(d))

(D 4) $le(d) = (epa = 0 \longrightarrow 0,\ le(rest(d,epa)) + 1)$
where: $epa = s\text{-}epa(top(d))$

(D 5) $depth(id,d) = le(rest(d,index(s(id,d),d)))$

(D 6) $disp_i = index(le(d)-i,d) \qquad \text{for } 0 \le i \le le(d) - 1$

(C 1) $D(update(e1,e2)) = D(e1) \cup D(e2)$

(C 2) $R(update(e1,e2)) \subseteq R(e1) \cup R(e2)$

(N 1) for $L1 = rest(L2)\ \&\ 1 \le i \le l(L1)$:

$rest(L1,i) = rest(L2,i)$

(L 1) for $1 \le i < l(d^n)$:

$\exists m(d^m = rest(d^n,i)\ \&\ 1 \le m < n)$

(L 2) $rest(d^n,1) = d^1$

(L 3) for $1 \le m < n$:
$U^m \subseteq U^n$

(L 4) for $1 \le m < n\ \&\ u \in U^m$:
$is\text{-}proc\text{-}den(u(dn^n)) \supset u(dn^n) = u(dn^m)$

(L 5) a) $R(e) \subseteq U$

b) for $u \in U\ \&\ is\text{-}proc\text{-}den(u(dn))$:
$R(s\text{-}e(u(dn))) \subseteq U$

(L 6) for $u \in R(e)\ \&\ is\text{-}proc\text{-}den(u(dn))$:

a) $1 \le s\text{-}epa(u(dn)) \le l(d)$

b) $s\text{-}e(u(dn)) = s\text{-}e(d_{s\text{-}epa(u(dn))})$

c) $s\text{-}tp(u(dn)) \Leftarrow s\text{-}tp(d_{s\text{-}epa(u(dn))})$

(L 7) for d such that $l(d) > 1$:
$1 \le epa < l(d)$

(L 8) for d such that $l(d) > 1$:
$e = update(s\text{-}e(d_{epa}),eo)$

(L 9) for $0 \leq i \leq le(d)$:

$$le(rest(d,index(i,d))) = le(d) - i$$

(L10) for d such that $l(d) > 1$:

$$tp \Longleftarrow s\text{-}tp(d_{epa})$$

(L11) $D(eo) = ID(tx,tp)$

<u>Theorem I</u>: for $id \in D(e)$:

$$id(e) = id(s\text{-}eo(d_{index(s(id,d),d)}))$$

<u>Theorem II</u>: for $id \in D(e)$:

$$id(e) = id(s\text{-}eo(d_{disp_{depth(id,d)}}))$$

REFERENCES

McCarthy, J. (1962), Towards a Mathematical Science of Computation, In: Information Processing, Popplewell C.M. (Ed.), Proc. 2nd IFIP Conf., München 1962, pp.21-28; North-Holland Publ.Com., Amerstdam, 1963.

Dijkstra, E.W. (1960), Recursive Programming, Numerische Mathematik 2, No.5.

Randell, B., Russell,L.J. (1964), ALGOL 60 Implementation, Academic Press, London, New York.

Walk, K. et al. (1969), Abstract Syntax and Interpretation of PL/I, IBM Laboratory Vienna, Techn.Report TR 25.098.

Lauer, P. (1968), Formal Definition of ALGOL 60, IBM Laboratory Vienna, Techn.Report TR 25.088.

Zimmermann,K. (1970), Loop Optimization: Representation and Proof of Correctness, IBM Laboratory Vienna, Techn.Report TR 25.108.

Henhapl,W. (1969), A Storage Model Derived from Axioms, IBM Laboratory Vienna, Techn.Report TR 25.100.

Henhapl,W., Jones,C.B. (1970), The Block Concept and some Possible Implementations, with Proofs of Equivalence, IBM Laboratory Vienna, Techn.Rep. TR 25.104.

Lucas,P. (1968), Two Constructive Realizations of the Block Concept and their Equivalence, IBM Laboratory Vienna, Techn.Report TR 25.085.

Lucas,P., Walk,K. (1969), On the Formal Description of PL/I, Annual Review in Automatic Programming, Vol.6, Part 3, Pergamon Press,

Chroust,G. (1970), Compilation of Expressions - A Bibliography, IBM Laboratory Vienna, Lab. Report, LR 25.3.061,(in preparation).

Van der Mey,G. (1962),Process for an ALGOL Translator, Report 164, Dr.Neher Laboratorium, Leidshendam.

Naur,P. (ed.) (1962), Revised Report on Algorithmic Language ALGOL 60, Comm.ACM Vol.6, No.1, pp.1-23.

ACKNOWLEDGEMENT

Much of the work done on the subject matter, but not represented in this paper, is due to Dr.W.Henhapl. The authors would like to take the opportunity of acknowledging the stimulus of their cooperation with him. The authors also gratefully acknowledge useful discussions with Dr.H.Bekić.