Verification of an Environment Management based on Operational Semantics for Static Scope Rules

MASAYUKI TAKEDA* and TAKUYA KATAYAMA*

This paper offers a different and simple approach to prove an environment management, i.e., a runtime system, for an ALGOL-like programming language with static scope rules. We formalize the static scope rules by starting with the concept of trace and define the central notion of environment, which represents all accessible data at a certain moment, in a more abstract way. Although correctness proofs of ALGOL-like runtime systems are a known subject, we arrive at a transparent correctness proof, which is easier to follow than the articles dealing with this subject, completely neglecting unnecessary technical details.

1. Introduction

This paper offers a different and simple approach to prove an environment management, i.e., a runtime system, for an ALGOL-like programming language with static scope rules. In general, correctness proofs of ALGOL-like runtime systems are a known subject, e.g., Jones & Lucas [2], McGowan [6], and Kandzia [5]. The proof of correctness in Jones & Lucas, for example, is based on the twin machine which combines two abstract machines, that is, the defining machine and the stack machine which define the language part and its implementation respectively. This paper, however, aims at a transparent proof completely neglecting unnecessary technical details. To this end, we develop the static scope rules by starting with the concept of trace and define the central notion of environment in a more abstract way.

Section 2 presents informal static scope semantics and introduces some restrictions of a language for the conciseness of discussions. In Sec. 3, we describe a program execution by giving a series of procedure activations and returns (called trace), and define the environment of a trace according to the static scope rules. Section 4 introduces the chain method which uses two chains (static link and dynamic link) for an environment management. In order to verify the correctness of the chain method, we show the correspondence between the environment defined in Sec. 3 and the one in this method (Theorem 2).

2. Static Scope Rules

In programming languages, there are two methods of communication between a called procedure and its caller; (1) global variable (with static/dynamic binding), (2) parameter passing (call by reference, by name, by value, by result, etc. [3]). We are concerned here with methods of global variable communication, since the central notion of an environment depends on it. The scope of a name can be classified into two large groups, that is to say, static and dynamic. In this paper, we consider the static scope rules which are widely used in ALGOL-like programming languages, such as ALGOL 60, PASCAL, or PL/I.

Static scope rules reflect the static nesting relations into their dynamic activations, and define the following; (1) all data which are accessible at a certain moment, and (2) legal procedure calls. An informal static scope semantics is described as follows in which some parameter restrictions are assumed for the conciseness of discussions.

[Parameter Restrictions]

We assume that parameters are absent or parameter passing has been performed except procedure parameters. The number of procedure parameters must be at most one, and the procedure which is able to call or refer to a formal procedure X is limited to the procedure which declares X as its formal parameter, that is, global formal procedures are not allowed.

[Static Scope Rules]

(R1) An environment of a procedure activation is a set of all variable locations which are accessible in this procedure and is defined as follows. Let data segment of a procedure be a set of its local variable locations. (a) If the activated procedure is a formal procedure and corresponding actual (not formal) procedure is P, then its environment is composed of its data segment occurrence and the most recently generated environment of the procedure activation, which declares P as its local procedure, at the time when P is transmitted for the first time. (b) If the activated procedure is not a formal one, then its environment is composed of its data segment occurrence and the most recently generated environment of the procedure activation which declares it at the time when it is called.

(R2) The procedure P may call or refer to the formal

*Department of Computer Science, Tokyo Institute of Technology.

procedure X only if there exists the actual parameter corresponding to X under the parameter restrictions. We assume that this rule R2 holds only in the case where the formal procedure X is eventually called.

(R3) An execution of a program begins with its main procedure, and a procedure Q (which is not a formal one) may be called and Q may be transmitted as a parameter within an activation of a procedure P only if all data segments of the ancestors of Q on the procedure declaration is contained in the environment of the activation of P.

Note that rule R1 shows the environment of a procedure activation and gives static scope to local variables. Rule R2 restricts an activation of a formal procedure and defines legal procedure parameter passing. Rule R3 defines an activation of a procedure (not formal one) and its transmission as a parameter, and gives static scope rules to procedure identifiers.

3. Formalization of Static Scope Rules

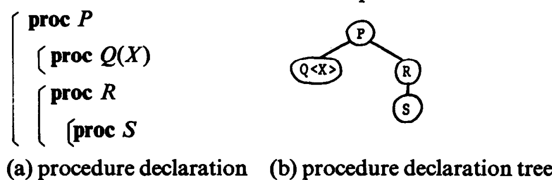
3.1 Description of Program Executions

In this section, we formalize the static scope rules shown above with the concept of trace as used in Hoare [1].

Definition 1. Procedure Declaration Tree T

Let $PNAME$ be a finite set of procedure names, and $FNAME$ be a finite set of formal procedure parameter names. Throughout this paper, these procedure names appear in upper case letters. A tree T is a *procedure declaration tree* if it consists of node P or $P\langle X\rangle$ for some $P \in PNAME$, $X \in FNAME$. We assume that procedure named P and formal procedure named X in T are all different.

Example 1.



(a) procedure declaration (b) procedure declaration tree

Definition 2. Legal Procedure Call Set $CALL$

(1) Let T be a procedure declaration tree. We take $Path(P)$ to be a path from the root node of T to the node whose procedure name is P, and describe it by a finite series of each procedure names.

(2) If H is a path in T, then *extension* $ext(H)$ is a set of procedure names whose distance from H are not more than one (See Fig. 1) [4]. It is formally defined as follows.

$$ext(H) = \{P \in PNAME \mid distance(H, P) \leq 1\} = \{P \in PNAME \mid P \in H \vee (\exists Q \in H \wedge branch(Q, P) \in T)\}$$

(3) If $P, Q \in PNAME$ and procedure call from P to Q is represented by $P \rightarrow Q$, then

$$CALL = \{P \rightarrow Q \mid P, Q \in PNAME \wedge Q \in ext(Path(P))\} \cup \{\rightarrow P \mid P \in PNAME \wedge P \textit{ is the root node of } T\}$$

is called the *legal procedure call set* on T. Note that $\rightarrow P$ represents the first procedure call in a program, that is, main procedure call.

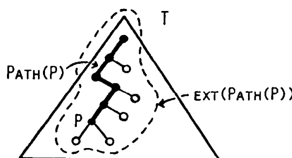


Fig. 1 Path and extension in a procedure declaration tree.

Definition 3. Trace, Program, Reduce Function

(1) We take $D_1 \times D_2 \times \cdots \times D_n$ to be the set of elements of the form $\langle x_1, x_2, \cdots, x_n\rangle$ where $x_i \in D_i$ $(1 \leq i \leq n)$, and if $x = \langle x_1, \cdots, x_n\rangle$ then $x \downarrow i$ denotes x_i $(1 \leq i \leq n)$.

We describe procedure activations and returns as follows. Let $ACT = (PNAME \cup FNAME) \times (PNAME \cup FNAME \cup \{\mathbf{null}\})$, then $p \in ACT$ denotes the *activation* of the procedure whose name is $p \downarrow 1$ and its actual parameter name is $p \downarrow 2$. Note that $p \downarrow 2 = \mathbf{null}$ means the activation without a parameter.

Let $RTN = \{\bar{p} \mid p \in ACT\}$, then $\bar{p}$ denotes the *return* from the activation p.

(2) Let $TR = (ACT \cup RTN)^*$, then we define a *trace* $e \in TR$ as a series of procedure activations and returns. Therefore, it is possible to describe a program execution by giving its trace, and each program is associated with a set of possible traces which are the possible series in the execution of its program. Let $TRACE \in 2^{TR}$ be a set of such traces, then a *program* is defined by the following quadruplet:

$program = (PNAME, FNAME, T, TRACE)$.

This association provides a foundation of a formal definitions of the language [1].

(3) Reduce Function r

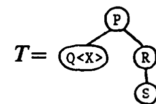
For any domain TR, a partial function r: $TR \rightarrow TR$ is defined by

1) $e_1, e_2 \in TR, p \in ACT \Rightarrow r(e_1 p \bar{p} e_2) = r(e_1 e_2)$

2) $e \in ACT^* \Rightarrow r(e) = e$.

This function reduces the legal pair of procedure activation and return in a trace. A trace $e \in TR$ is *reducible* if $r(e) \in ACT^*$.

Example 2. $T =$



$PNAME = \{P, Q, R, S\}$, $FNAME = \{X\}$,

If an activation of a procedure is represented by its lower case letter, then the following trace is reducible.

$$e = p\ r\ s\ \bar{s}\ \bar{r}\ r\ q\ x \quad (S)$$

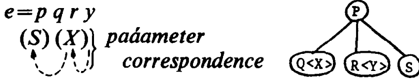
$$r(e) = p\ r\ q\ x \quad (S)$$

where if an actual parameter exists in an activation, it is shown in the parenthesis; e.g., $p{\downarrow}2 = \mathbf{null}$, $p{\downarrow}2 = S$.

Definition 4. Parameter Matching Function *Match*

(1) If a procedure P is passed as a parameter, then the procedure which transmits P for the first time is called the *base procedure* of P.

Example 3.



The procedure P in this trace is the base procedure of S and X, and the actual procedure name ($\in PNAME$) corresponding to the formal procedure X and Y is S.

(2) To make a formal parameter correspond with an actual parameter, we define new domain *SEG* for each activation in a reduced trace as follows.

$$SEG = PNAME \times D \times N,$$

where *PNAME* part shows the activated procedure name. D is a set of data segments (see static scope rule R1) and represents its data segment occurrence. N is a non-negative integer and this part points to the previous procedure activation which way fram the environment in a reduced trace and it is called the *access link.*

(3) Parameter matching function *Match*: $ACT^* \to SEG^*$ is defined by

$Match(\varepsilon) = \varepsilon$, and

for any $\mathbf{p} \in ACT^*$, $q \in ACT$,

$$Match(\mathbf{p}q) = Match(\mathbf{p}) \circ m(\mathbf{p}, q{\downarrow}1),$$

where m: $ACT^* \times (PNAME \cup FNAME) \to SEG$ is a partial function defined as follows.

If $\mathbf{p} \in ACT^*$, $Q \in (PNAME \cup FNAME)$ then

$$m(\mathbf{p}, Q) = \begin{cases} Q \in PNAME \Rightarrow \langle Q, dseg(Q), |\mathbf{p}| \rangle, & (3.1) \\ Q \in FNAME \wedge last(\mathbf{p}){\downarrow}2 \in (PNAME \cup FNAME) \wedge \\ Q \text{ is the formal parameter of } last(Match(\mathbf{p})){\downarrow}1 \\ \quad \Rightarrow m(\mathbf{p}/last(\mathbf{p}), last(\mathbf{p}){\downarrow}2), & (3.2) \\ \text{elase} \Rightarrow \bot. & (3.3) \end{cases}$$

In this definition, we introduce the following notations:

$\circ$: concatenation,
$dseg(Q)$: data segment of the procedure Q,
$|\mathbf{p}|$: length of series $\mathbf{p}$,
$/$: right quotient,
$last(\mathbf{p})$: last (right most) element of $\mathbf{p}$,
$\bot$: undefined value.

For each procedure activation in a reduced trace, the function m creates a *SEG* value which is composed of its procedure name ($\in PNAME$), data segment occurrence, and access link (3. 1). In case of a formal procedure activation (3. 2), corresponding actual parameter ($last(\mathbf{p}){\downarrow}2$) is applied to m, and resulting access link points to its base procedure activation. If a procedure is activated directly, its access link points to its calling procedure activation. If there is no actual parameter or illegal parameter accessing exists (3. 3), then m returns $\bot$ as its value.

Example 4.

For the trace shown in Example 2, parameter matching can be performed as follows.

$$Match(r(e)) = p'\ r'\ q'\ s' \quad \text{(0)(1)(2)(2)} \text{ illustration of access link}$$

where p' shows the *SEG* value whose first element (procedure name) is represented by its upper case letter P, and the number with an arrow below it illustrates its access link.

In the following, we define the conditions of a trace which ensure the legal procedure activation according to the static scope rules described in Sec. 2.

Definition 5. Feasible Trace

A trace $e \in TR$ is *feasible* if the following conditions hold.

(1) e is reducible.

(2) For any prefix e_1 of e such that $r(e_1) \neq \varepsilon$,

$$\exists u_1 u_2 \cdots u_n \in SEG^*[Match(r(e_1)) = u_1 u_2 \cdots u_n \wedge \forall j[2 \leq j \leq n \Rightarrow \exists i[i = u_j{\downarrow}3 \wedge 1 \leq i \leq j-1 \wedge (u_i{\downarrow}1 \to u_j{\downarrow}1) \in CALL]] \wedge (\to u_1{\downarrow}1) \in CALL].$$

Condition (1) ensures the legal pair of procedure activation and return in a trace. Condition (2) shows that the procedure call from $u_i{\downarrow}1$ to $u_j{\downarrow}1$, which have the following relation by an access link, is legal,

$$Match(r(e_1)) = \cdots u_i \cdots u_j \cdots \quad \text{access link,}$$

and this condition represents the static scope rule R3. The rule R2 is represented by the parameter matching function Match (Def. 4–(3)).

Lemma 1.

A trace which is a prefix of some feasible trace is also feasible.

3.2 Environment of Trace

In this section, we define the environment of a trace according to the static scope rules, and investigate the condition of the trace whose environment is well-defined (Theorem 1).

In order to pick up the procedure activations which are

chained by access links, we introduce the function *Pos* which returns the set of these procedure positions in a reduced trace.

Definition 6. *Pos*: $SEG^* \to 2^N$
For any $u \in SEG^*$,

$$Pos(\mathbf{u}) = \begin{cases} |\mathbf{u}| = 0 \Rightarrow \phi: \text{empty set}, \\ \text{else} \Rightarrow \{|\mathbf{u}|\} \cup Pos(first(\mathbf{u}, last(\mathbf{u})\downarrow 3)) \end{cases}$$

where $first(\mathbf{u}, n)$ gives the series which consists of first n elements of $\mathbf{u}$.

Example 5.
Let $\mathbf{u} = \boxed{u_1}\ u_2\ \boxed{u_3}\ u_4\ \boxed{u_5}$
(0) (1) (1) (3) (3)

then $Pos(\mathbf{u}) = \{5\} \cup Pos(u_1u_2u_3)$
$= \{5\} \cup \{3\} \cup Pos(u_1)$
$= \{5\} \cup \{3\} \cup \{1\} \cup Pos(\varepsilon)$
$= \{1, 3, 5\}$

The environment of a trace e is defined by both $Path(P)$ and $Match(r(e))$ where P is the most recently activated procedure without the corresponding return in a trace. $Path(P)$ is used for giving a static nesting relation of a procedure declaration with respect to a trace e. The following function *Select* selects the *SEG* values, whose data segment parts form the environment of a trace e, from $Match(r(e))$.

Definition 7. *Select*: $PNAME^* \times SEG^* \to SEG^*$
Let $H = P_1P_2 \cdots P_t \quad (P_i \in PNAME,\ 1 \le i \le t)$
$\mathbf{u} = u_1u_2 \cdots u_n \quad (u_i \in SEG,\ 1 \le i \le n,\ u_n\downarrow 1 = P_t)$,
then
$Select(H, \mathbf{u}) = v_1v_2 \cdots v_t (v_i \in SEG,\ 1 \le i \le t)$
where,

$$\begin{cases} v_t = u_n; \\ \text{for any } 1 \le i \le t-1 \\ \quad \exists k \in Pos(\mathbf{u})[v_i = u_k \wedge v_i\downarrow 1 = P_i \wedge \\ \quad\quad \forall j \in Pos(\mathbf{u})[k < j \le v_{i+1}\downarrow 3 \Rightarrow u_j\downarrow 1 \ne P_i]]. \end{cases}$$

Definition 8. Environment of trace e: $env(e)$
For any trace $e \in TR$, the partial function env: $TR \to D^*$ is defined by

$$env(e) = Select(Path(P), Match(r(e)))\downarrow 2,$$

where $P = last\ (Match(r(e)))\downarrow 1 \in PNAME$, and the operator $\downarrow$ operates on all elements of the series. If $|r(e)| = 0$ then $env(e) = \varepsilon$.

Example 6. The environment of the trace shown in Example 2 is the following.

$Match(r(e)) = p'r'q's'$

$env(e) = Select(Path(S), p'r'q's')\downarrow 2$
$= Select(PRS, p'r'q's')\downarrow 2$
$= (p'r's')\downarrow 2.$

The following theorem shows that the property of feasibility ensures not only the legal procedure activation but also the well-definedness of the environment of a trace.

Theorem 1.
If e is a feasible trace then env(e) is well-defined.
Proof. By induction on $|r(e)|$.
1. The result obviously holds for $|r(e)| = 0$, and 1.
2. Assume the result holds for $|r(e)| \le n$.

For the case of $|r(e)| = n+1$, if e is a feasible trace then

$$Match(r(e)) = u_1u_2 \cdots u_{n+1} \in SEG^*$$

is well-defined, and let $Q = u_{n+1}\downarrow 1$ then by Definition 5–(2)

$$\exists i[i = u_{n+1}\downarrow 3 \wedge 1 \le i \le n \wedge P \to Q \in CALL]$$

where $P = u_i\downarrow 1$.

$$Match(r(e)) = \cdots \overset{\exists\ P}{u_i} \cdots \overset{Q}{u_{n+1}}$$

Since there exists the prefix trace e_1 of e such that

$$Match(r(e_1)) = u_1u_2 \cdots u_i \quad \text{(by Lemma 1)},$$

then

$$env(e_1) = Select(Path(P), Match(r(e_1)))\downarrow 2$$

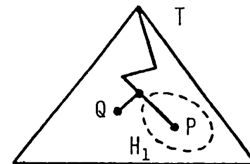
is well-defined by induction hypothesis.
Furthermore, since $P \to Q \in CALL$,

$$\exists H_1[Path(Q) = (Path(P)/H_1)\circ Q],$$

and

$$env(e) = (Select(Path(P), Match(r(e_1)))\downarrow 2/\rho)\circ u_{n+1}\downarrow 2$$
$$= (env(e_1)/\rho)\circ u_{n+1}\downarrow 2,$$

where ρ is some suffix of $env(e_1)$.
Therefore $env(e)$ is well-defined.



□

Beginning with the next section, we present some properties with respect to the environment $env(e)$. The proofs are omitted for want of space.

Lemma 2.
For any activation $p \in ACT$, and trace $e_1, e_2 \in TR$ such that $e_1p\bar{p}e_2$ is feasible, then $env(e_1p\bar{p}e_2) = env(e_1e_2)$.
Lemma 2 shows that the legal pair of activation and

return in a feasible trace do not affect its environment.

Lemma 3.

For any trace $e \in TR$, and formal procedure activation $q \in ACT$, $q{\downarrow}1 \in FNAME$, if e_1 is the prefix of the feasible trace eq such that

$$Match(r(eq)) = u_1 \cdots u_i \cdots u_n,\ u_n{\downarrow}3 = i, \text{ and}$$

$$Match(r(e_1)) = u_1 \cdots u_i,$$

then

$$env(eq) = env(e_1 p),$$

where $p \in ACT$ is the activation of the procedure $u_n{\downarrow}1 \in PNAME$.

Lemma 3 shows that the activation of a formal procedure restores the environment in which the corresponding actual procedure ($\in PNAME$) is transmitted for the first time. Obviously, Lemma 3 holds for $q{\downarrow}1 \in PNAME$.

Lemma 4.

If $e \in TR$ is the feasible trace such that

$$Match(r(e)) = u_1 \cdots u_i \cdots u_n, \quad u_n{\downarrow}3 = i$$

$$r(e) = p_1 \cdots p_i \cdots p_n, \quad p_n{\downarrow}1 \in FNAME,$$

then $p_i{\downarrow}1$ is the base procedure of the formal procedure $p_n{\downarrow}1$ and the corresponding actual procedure name ($\in PNAME$) is $u_n{\downarrow}1$.

4. Verification of an Environment Management

The method of an environment management with two chains (static and dynamic links), what is called chain method or display method, is widely used in ALGOL-like language processors [7]. Although correctness proofs of such runtime systems are a known subject [2], we can show a transparent correctness proof, which is easier to follow than the articles dealing with this subject, avoiding technical details.

In Sec. 4.1, we present the chain method under the parameter restrictions shown in Sec. 2, and define the environment $env_\omega(e)$ of the chain method in Sec. 4.2. We then prove the equivalence of two environments, that is $env(e)$ and $env_\omega(e)$, in Sec. 4.3.

4.1 Chain Method

A *stack*, which keeps the values of local variables of a procedure and other control informations for an environment management, is composed of the following elements:

1. *static link*$\cdots$set of pointers which indicate the origin addresses of the accessible stack elements and keep the order of its static nesting in a program,
2. *dynamic link*$\cdots$pointer which keeps the order of the dynamic generation of a stack element,
3. *level*$\cdots$static nesting level of a procedure,
4. *parameter information*$\cdots$it is composed of an actual parameter name and a pointer to the origin address of its base procedure stack element,
5. *data segment*$\cdots$set of local variables of a procedure.

These structures are shown in a PASCAL-like programming language as follows.

```
const maxlev = maximum proc nesting level;
type stack = stack of rec;
  rec = record
          disp  : array [1..maxlev] of ↑rec;
          rtn   : ↑rec;
          level : 1..maxlev;
          para  : para info;
          dseg  : data segment
        end;
  para info = record
            name: proc name {∈ PNAME};
            base : ↑rec
            end;
  data segment = set of local variable;
var S: stack;
    A: ↑rec {pointer to current stack element}
```

The following functions are prepared for the management of the stack S.

push(S, P)—pushes the new stack element for the procedure $P \in PNAME$ into stack S and returns the resulting stack as its value.

pop(S)—pops the top element from stack S and returns the remainder stack.

Definition 9. Initial state and state transitions

(1) Initial state.

$$S = \phi\text{: empty stack;}$$

$$A = \mathbf{nil}$$

(2) Activation of a procedure P with an actual parameter Q, where $P \in PNAME \cup FNAME$, $Q \in PNAME \cup FNAME \cup \{\mathbf{null}\}$.

We indicate the consequent states by giving apostrophes.

$$P_a = \begin{cases} P & (P \in PNAME), \\ A{\uparrow}.para.name & (P \in FNAME); \end{cases}$$

Note that P_a is an actual procedure name $\in PNAME$ of P.

$$S' = push(S, P_a);$$

$$A'{\uparrow}.dseg = dseg(P_a);$$

$$A'{\uparrow}.rtn{\uparrow} = A{\uparrow};$$

$$A'{\uparrow}.level = level(P_a);$$

where $level(P)$ gives the static nesting level of a procedure $P \in PNAME$.

$$A'\uparrow.disp[i]\uparrow = \begin{cases} A\uparrow.disp[i]\uparrow & (1 \le i < A'\uparrow.level \wedge P \in PNAME), \\ A\uparrow.para.base\uparrow.disp[i]\uparrow & (1 \le i < A'\uparrow.level \wedge P \in FNAME), \\ A'\uparrow & (i = A'\uparrow.level), \\ \perp & (A'\uparrow.level < i \le maxlev); \end{cases}$$

$$A'\uparrow.para.name = \begin{cases} A\uparrow.para.name & (Q \in FNAME), \\ Q & (Q \in PNAME), \\ \perp & (Q = \textbf{null}); \end{cases}$$

$$A'\uparrow.para.base\uparrow = \begin{cases} A\uparrow.para.base\uparrow & (Q \in FNAME), \\ A\uparrow & (Q \in PNAME), \\ \perp & (Q = \textbf{null}) \end{cases}$$

(3) Return.

$$A'\uparrow = A\uparrow.rtn\uparrow;$$
$$S' = pop(S)$$

Example 7.

For the procedure declaration tree shown in Example 2, the stack of the trace $e = pr\underset{(S)}{q}x$ is illustrated in Fig. 2.

4.2 Environment on Chain Method

Before defining the environment of the chain method, we represent a stack of a trace by the following function.

Definition 10. Stack of trace e: $stack(e)$

A stack of a trace $e \in TR$ is defined as

$$stack(e) = A\uparrow = \langle A\uparrow.disp[1]\uparrow, \cdots, A\uparrow.disp[maxlev]\uparrow, A\uparrow.rtn\uparrow, A\uparrow.level, A\uparrow.para, A\uparrow.dseg\rangle,$$

where A points to the current stack element based on e.

Lemma 5.

For any activation $p \in ACT$, and trace $e_1, e_2 \in TR$ such that $e_1 p\bar{p} e_2$ is feasible, then $stack(e_1 p\bar{p} e_2) = stack(e_1 e_2)$.

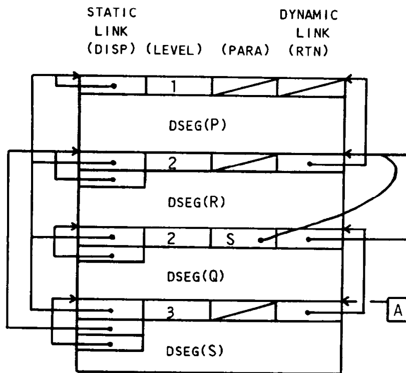


Fig. 2 Linkage of stack elements ($e = p\ r\ \underset{(S)}{q}\ x$).

Definition 11. Environment

The environment $env_\omega(e)$ of a trace e on chain method is defined by

$$env_\omega(e) = \begin{cases} A\uparrow.disp[1]\uparrow.dseg\circ \\ \vdots \\ A\uparrow.disp[A\uparrow.level]\uparrow.dseg, \end{cases}$$

where $A\uparrow = stack(e)$.

The next lemma corresponds to Lemma 2.

Lemma 6.

For any activation $p \in ACT$, and trace $e_1, e_2 \in TR$ such that $e_1 p\bar{p} e_2$ is feasible, then $env_\omega(e_1 p\bar{p} e_2) = env_\omega(e_1 e_2)$.

We present some properties of a stack with respect to an activation of a formal procedure.

Lemma 7.

For any trace $e \in TR$ and formal procedure activation $q \in ACT$, $q\downarrow 1 \in FANME$, if eq is feasible and $A\uparrow = stack(eq)$ then the stack element of the base procedure of $q\downarrow 1$ is $A\uparrow.rtn\uparrow.para.base\uparrow$, and the corresponding actual procedure name ($\in PNAME$) is $A\uparrow.rtn\uparrow.para.name$.

Lemma 8.

For any trace $e \in TR$ and formal procedure activation $q \in ACT$, $q\downarrow 1 \in FNAME$, if e_1 is the prefix of the feasible trace eq such that

$$Match(r(eq)) = u_1 \cdots u_i \cdots u_n,\ u_n\downarrow 3 = i, \text{ and}$$

$$Match(r(e_1)) = u_1 \cdots u_i,$$

then

$$stack(eq).rtn\uparrow.para.base\uparrow = stack(e_1).$$

Lemma 8 shows the relation between the access link of a formal procedure activation and the para.base link of its calling procedure on chain method.

The next lemma corresponds to Lemma 3.

Lemma 9.

For any trace $e \in TR$ and formal procedure activation $q \in ACT$, $q\downarrow 1 \in FNAME$, if e_1 is the prefix of the feasible trace eq such that

$$Match(r(eq)) = u_1 \cdots u_i \cdots u_n,\ u_n\downarrow 3 = i, \text{ and}$$

$$Match(r(e_1)) = u_1 \cdots u_i,$$

then

$$env_\omega(eq) = env_\omega(e_1 p),$$

where $p \in ACT$ is the activation of the procedure $u_n\downarrow 1 \in PNAME$.

4.3 Correctness Proof of Chain Method

Theorem 2 given below shows the correspondence of

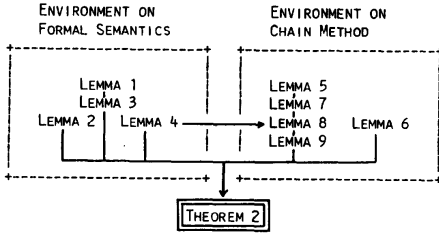


Fig. 3 Correctness proof of chain method.

two environments, that is, $env(e)$ and $env_\omega(e)$, and proves the correctness of the chain method. Figure 3 shows the dependency relation of each lemma used in this theorem.

Theorem 2. *For any feasible trace* $e \in TR$,

$$env_\omega(e) = env(e).$$

Proof. By induction on $|e|$.

1. $(|e|=0)$

$$env_\omega(e) = \varepsilon = env(e) \quad (\text{since } A = \mathbf{nil}).$$

2. $(|e|=1)$

If $p \in ACT$ is a feasible trace then its procedure name $p\downarrow 1$ is in $PNAME$ and

$$\begin{aligned} env_\omega(p) &= A\uparrow.dseg \quad \text{where } A\uparrow = stack(p) \\ &= dseg(P) \quad \text{where } P = p\downarrow 1 \\ &= env(p). \end{aligned}$$

3. Assume the result holds for $|e| \le k$.

3–1. For the case of $e\bar{q}$ where $\bar{q} \in RTN$, $|e|=k$, if $e\bar{q}$ is a feasible trace then

$$\exists p \in ACT, \exists e_1, e_2 \in TR[e\bar{q} = e_1 p\bar{p} e_2 \wedge |e_1 e_2| = k-1].$$

Therefore

$$\begin{aligned} env_\omega(e\bar{q}) &= env_\omega(e_1 p\bar{p} e_2) \\ &= env_\omega(e_1 e_2) \quad \text{(Lemma 6)} \\ &= env(e_1 e_2) \quad \text{(hypothesis)} \\ &= env(e_1 p\bar{p} e_2) \quad \text{(Lemma 2)} \\ &= env(e\bar{q}). \end{aligned}$$

3–2. For the case of eq where $q \in ACT$, $|e|=k$.

3–2–1. If $e = e_0\bar{p}$ ($e_0 \in TR$, $\bar{p} \in RTN$) then in a similar way of 3–1 $env_\omega(eq) = env(eq)$.

3–2–2. Let $e = e_0 p$ ($e_0 \in TR$, $p \in ACT$).

a) If $q\downarrow 1 \in PNAME$ then

$$\begin{aligned} env_\omega(eq) &= \begin{cases} A'\uparrow.disp[1]\uparrow.dseg\circ \\ \vdots \\ A'\uparrow.disp[A\uparrow.level]\uparrow.dseg \end{cases} \\ &\qquad \text{where } A'\uparrow = stack(eq) \\ &= \begin{cases} A'.disp[1]\uparrow.dseg\circ \\ \vdots \\ A\uparrow.disp[M-1]\uparrow.dseg\circ \\ \quad dseg(Q) \end{cases} \\ &\qquad \text{where } A\uparrow = stack(e),\ Q = q\downarrow 1,\ M = level(Q). \end{aligned}$$

If eq is a feasible trace and $Match(r(eq)) = u_1 \cdots u_{n-1} u_n \in SEG^*$, then $u_n\downarrow 3 = n-1$ by Definition 4–(3), and $(u_{n-1}\downarrow 1 \to u_n\downarrow 1) \in CALL$ by Definition 5–(2). That is, if $P = u_{n-1}\downarrow 1$ and $L = level(P)$ then $P \to Q$ is a legal procedure call and $M-1 \le L$.

Therefore, there exists some suffix $\rho \in D^*$ of $env_\omega(e)$ and

$$\begin{aligned} env_\omega(eq) &= (env_\omega(e)/\rho)\circ Q \\ &= (env(e)/\rho)\circ Q \quad \text{(hypothesis)} \\ &= env(eq). \end{aligned}$$

b) If eq is a feasible trace, $q\downarrow 1 \in FNAME$, and $Match(r(eq)) = u_1 \cdots u_n \in SEG^*$, then by Definition 5–(2)

$$\exists i[i = u_n\downarrow 3 \wedge 1 \le i \le n-1 \wedge (u_i\downarrow 1 \to u_n\downarrow 1) \in CALL].$$

Let e_1 be the prefix of eq such that

$$Match(r(e_1)) = u_1 \cdots u_i,$$

then by Lemma 9

$$env_\omega(eq) = env_\omega(e_1 p_a),$$

where p_a is the activation of the procedure $u_n\downarrow 1$.

Since $|e_1 p_a| \le k$,

$$\begin{aligned} env_\omega(e_1 p_a) &= env(e_1 p_a) \quad \text{(hypothesis)} \\ &= env(eq) \quad \text{(Lemma 3)}. \end{aligned}$$

Therefore

$$env_\omega(eq) = env(eq). \qquad \square$$

5. Conclusion

We develop the static scope rules by starting with the concept of trace and define the central notion of environment, which represents all accessible data at a certain moment, in a more abstract way. Although correctness proofs of ALGOL-like runtime systems are a known subject, we arrive at a transparent correctness proof, which is easier to follow than the articles dealing with this subject, completely neglecting unnecessary technical details.

In this paper, we consider the *procedure entry & static scope rules* which is used in PASCAL. This method is, however, applicable to *block entry & static scope rules* used in ALGOL60 or PL/I by regarding a trace as a series of block activations and returns.

We assume parameter restrictions for the conciseness of discussions. This restrictions may be removed if (1) list representation is used for procedure parameters and (2) an environment consists of both data segments and parameter passing informations. Detailed discussions for this extension will be the subject of a future paper.

Acknowledgement.

We wish to thank Professor Hajime Enomoto for his valuable advice, and Naoki Yonezaki and Toshio Miyachi for carefully reading earlier drafts of this paper.

References

1. Hoare, C. A. R. Some Properties of Predicate Transformers,

JACM, **25**, No. 3, 461–480 (July 1978).

2. Jones, C. B., Lucas, P. Proving Correctness of Implementation Techniques, Symp. on Semantics of Algorithmic Languages (E. Engeler, ed), *Lecture Notes in Mathematics* 188, 178–211, Springer (1971).

3. Jones, N. D., Muchnick, S. S. TEMPO: A Unified Treatment of Binding Time and Parameter Passing Concepts in Programming Languages, *Lecture Notes in Computer Science*, No. 66, Springer-Verlag, Berlin (1978).

4. Katayama, T., Takeda, M. Correctness Proof for an Implementation of the Scope Rule, Proc. IECE symp. on Automata and Language AL 79–23, 31–37 (1979).

5. Kandzia, P. On the Most Recent Property of ALGOL-like Programs, *Lecture Notes in Computer Science* 14, 97–111, Springer (1974).

6. McGowan, C. L. The 'most recent' Error: Its Causes and Correction, *SIGPLAN Notices* 7, No. 1, 191–202, (1972).

7. Wirth, N. The Design of a PASCAL Compiler, *Software-Practice and Experience*, **1**, No. 4, 309–333 (October 1971).

(Received May 7, 1981: Revised October 26, 1981)