Proving the Correctness of the Static Link Technique Using Evolving Algebras

Markus Mohnen
RWTH Aachen, Lehrstuhl für Informatik II
Ahornstraße 55, 52056 Aachen, Germany
email: `markusm@zeus.informatik.rwth-aachen.de`

Abstract

The *static link technique* is used for the stack-based implementation of imperative programming languages which admit nested recursive procedure declarations. It was suggested in 1960 by Dijkstra as an extension of the stack technique for the evaluation of expressions. Its basic idea is to access global variables by tracing a static link chain to lower stack elements. *Evolving algebras* are a new method for the definition of operational semantics of abstract machines. Based on an appropriate stack machine, defined as an evolving algebra, and a functional description of a compiler for a sample language, we will give a complete proof of this technique. Especially, we will transform the evolving algebra for the stack machine into a more abstract one, i.e., into a new machine which is closer to the denotational semantics of the sample language.

Introduction

The basic concept of imperative programming languages is the assignment of values to variables. As a conclusion it is crucial for the implementation how variables are accessed. The implementation is easy as long as one of these two conditions holds:

(i) no recursion (`FORTRAN77`)

(ii) no nested procedure declarations (`C`)

In the first case, each variable in the program gets a statically allocated memory cell. Access to a variable will access the unique memory cell for this variable. If there is a second call to a procedure `P`, assignments to the local variables of `P` during the execution of the second instance will overwrite the values of the local variables of the first instance of `P`.

In the second case, the variables are divided into two classes: the first class holds the globally declared variables, i.e. the variables which are declared *outside* of all procedures. These variables get a statically allocated memory cell. The second class holds the other variables, i.e. the variables, which are declared *inside* a procedure `P`. For each instance of `P` during execution there will be a new allocated set of memory cells for the local variables of `P` on a *stack*. This set is part of the *activation record* for this instance of `P`. It will be removed as soon as the instance of the procedure terminates. The procedure can only access the global variables and the variables declared within `P`. Access to a global variable is handled in the same way as in the first case. On the other hand, access to local variables will access the last allocated cell for this variable on the stack. On termination of a recursive call to `P`, the values of the previous instance of `P` will be restored.

The implementation gets more complicated when none of the above conditions is fulfilled. These languages allow nested procedure declarations and recursion (like, for instance, `PASCAL`). Now, a procedure `P` can access all variables declared in `P` or its surrounding procedures. The stack mechanism can handle the recursion, but the access to those variables which are *not* declared in `P` causes some problems.

The solution is the *static link technique*. It extends the activation records by pointers (*static links*) to activation records on inner stack positions. A variable is accessed by tracing the static links up to the latest incarnation of this variable on the stack.

Moreover, the static links technique solves the problem of name clashes in a reasonable way. A procedure `P` can declare a new variable `X`. A problem arises, as soon as a call to `P` is performed and there is already a variable `X` at this time. Which `X` should be accessed in `P`? With *dynamic scope semantics*, the old `X` is used; this, of course, makes programs very unclear, because the semantics of `P` does not only depend on `P`, but also on the *context* in which `P` is embedded. The *static scope semantics*, which is implemented by the static link technique, uses the `X` which is declared in `P`. This, of course, is what a programmer expects to be the semantics of the access to `X`.

Evolving algebras are a new concept for the description of operational semantics and abstract machines ([Gur88],[Gur91]). Its basic idea is to describe the behavior of the machine by a transition system with algebras as states. Each state represents the complete state of the computation. The use of evolving algebras has several advantages:

- the usual definitions of machine state and transition relation can be omitted, because they are already fixed,
- the comparison of different machines is easier,
- the technique of *refinement* is a natural approach, which is close to the actual development process of an abstract machine. This technique is supported in a canonical way by the notion of *equivalence* of evolving algebras, and
- the description of the dynamic behavior of a machine in shorter because only those components of the state of the machine must be specified which are affected in a certain situation.

There has been work in the area of the block concept and its correctness. But most of it was concerned with the display technique ([JL71, MS76, Pol81]), which is much easer to proof[1]. A proof of the static-link technique with respect to an operational semantic can be

[1]The reason for this is that variables and linking-information are stored in different parts of the storage. Activation record are not linked, but there is a *display* which contains a direct link to the appropriate activation record for each possible level. As a consequence the structure is correct by definition.

found in [HJ70], where the 'Twin-Machine' technique is used. The basic idea is to construct a machine with two independent mechanisms of storage access, representing the operational semantics resp. the static-link technique and to prove that both are equivalent in each state of the machine.

In this paper, we will present a proof for the correctness of the static-link technique with respect to a denotational semantics in the frame of a subset of PASCAL. The language is simple enough to allow the proof, but not trivial. The proof is based on the functional compiler description in [Ind88].

The paper is organized in the following way: Section 1 introduces the source language PLP by giving its abstract syntax and denotational semantics. The next section introduces evolving algebras and Section 3 gives the description of the target machine as an evolving algebra. The fourth section describes the compilation process via the translation mapping.

The access to the variables on the procedure stack of the machine is controlled by the 'base'-function. The fifth section is dedicated to this 'base'-function. We will show that the execution of a compiled program preserves a certain structure of the procedure stack.

We exploit this structure in Section 6, where we define a modified machine. We change the access to variables in order to get closer to the one used in the denotational semantics. Section 7 will present a proof of the equivalence of the original and the modified machine.

In the final Section 8 we prove the equivalence of the denotational semantics of PLP and the implementation by the modified machine.

The appendix contains some notations in this paper.

1 The source language PLP

We define PLP by its abstract syntax. Of course, we do not bother with the problem of parsing, so we can define denotational semantics and translation mapping by means of structural induction.

<u>Int</u>	z	$z \in \mathbb{Z}$
<u>VIde</u>	$\mathtt{X}_{l,o}$	$l \in \mathbb{N}_0,\ o \in \mathbb{N}$
<u>PIde</u>	$\mathtt{P}_{l,o}$	$l \in \mathbb{N}_0,\ o \in \mathbb{N}$
<u>PLP-Exp</u>	$E ::= z \mid \mathtt{X}_{l,o} \mid E_1 + E_2$	
<u>PLP-Decl</u>	$\Delta ::= \Delta_v \Delta_p$	
	$\Delta_v ::= \varepsilon \mid \mathtt{var}\ \mathtt{X}_{l,1}, \ldots, \mathtt{X}_{l,n};$	
	$\Delta_p ::= \varepsilon \mid \mathtt{proc}\ \mathtt{P}_{l,1}; B_1; \ldots; \mathtt{P}_{l,n}; B_n;$	
<u>PLP-Cmd</u>	$\Gamma ::= \mathtt{X}_{l,o} := E \mid$	
	$\mathtt{P}_{l,o}() \mid$	
	$\mathtt{if}\ E\ \mathtt{then}\ \Gamma_1\ \mathtt{else}\ \Gamma_2 \mid$	
	$\mathtt{while}\ E\ \mathtt{do}\ \Gamma \mid$	
	$\mathtt{begin}\ \Gamma_1; \ldots; \Gamma_n\ \mathtt{end}$	
<u>PLP-Block</u>	$B ::= \Delta\Gamma$	
<u>PLP-Prog</u>	$P ::= \mathtt{in/out}\ \mathtt{X}_{0,1}, \ldots, \mathtt{X}_{0,n}; B.$	

As additional conditions, we demand that

(i) the first index of a variable or procedure identifier is the nesting level of its declaration, and

(ii) a variable is declared before it is used.

The denotational semantics will be undefined if one of these conditions is violated.

The naming restrictions will enable us to drop the concept of *locations*, which is needed to ensure the *static scope semantics* for variables. In fact, dynamic scope and static scope are the same in this case. One can prove, that this semantics delivers the same results as the semantics given in [Ind88].

First, we have to define the semantic domains, on which the semantic functions operate:

- $\underline{\mathrm{S}} := \{\chi \mid\mid \chi : \underline{\mathtt{VIde}} \dashrightarrow \mathbb{Z}\}$ the set of *states*, which hold the values of variables.
  As a special notation, we write χ_l for the variable binding, which is created by restricting χ to the set $\{\mathtt{X}_{l,n} \mid\mid n \in \mathbb{N}\}$.
- $\underline{\mathrm{C}} := \{\Phi \mid\mid \Phi : \underline{\mathrm{S}} \dashrightarrow \underline{\mathrm{S}}\}$ the set of *state transformations*, used as values of procedures.
- $\underline{\mathrm{U}} := \{\pi \mid\mid \pi : \underline{\mathtt{PIde}} \dashrightarrow \underline{\mathrm{C}}\}$ the set of *environments*, which assign a state transformation to a procedure identifier.

Now, we give semantic functions for each component of a PLP-program:

- $\mathfrak{E} : \underline{\mathtt{PLP\text{-}Exp}} \times \underline{\mathrm{S}} \dashrightarrow \mathbb{Z}$

$$\begin{array}{rcl}
\mathfrak{E}[\![z]\!](\chi) & := & z \\
\mathfrak{E}[\![\mathtt{X}_{l,o}]\!](\chi) & := & \chi(\mathtt{X}_{l,o}) \\
\mathfrak{E}[\![E_1 + E_2]\!](\chi) & := & \mathfrak{E}[\![E_1]\!](\chi) + \mathfrak{E}[\![E_2]\!](\chi)
\end{array}$$

- $\mathfrak{D} : \underline{\mathtt{PLP\text{-}Decl}} \times \mathbb{N}_0 \times \underline{\mathrm{U}} \times \underline{\mathrm{S}} \dashrightarrow \underline{\mathrm{U}} \times \underline{\mathrm{S}}$

$$\begin{array}{rcl}
\mathfrak{D}[\![\Delta_v \Delta_p]\!](l, \pi, \chi) & := & \mathfrak{D}[\![\Delta_p]\!](l, (\mathfrak{D}[\![\Delta_v]\!](l, \pi, \chi))) \\
\mathfrak{D}[\![\varepsilon]\!](l, \pi, \chi) & := & (\pi, \chi) \\
\mathfrak{D}[\![\mathtt{var}\ \mathtt{X}_{l,1}, \ldots, \mathtt{X}_{l,n};]\!](l, \pi, \chi) & := & (\pi, \chi[\mathtt{X}_{l,1}/0, \ldots, \mathtt{X}_{l,n}/0]), \\
\mathfrak{D}[\![\mathtt{proc}\ \mathtt{P}_{l,1}; B_1; \ldots; \mathtt{P}_{l,n}; B_n;]\!](l, \pi, \chi) & := & (\pi[\mathtt{P}_{l,1}/\Phi_1, \ldots, \mathtt{P}_{l,n}/\Phi_n], \chi), \\
 & & \text{where } (\Phi_1, \ldots, \Phi_n) \text{ is the least fixpoint of} \\
 & & \Phi_i = \mathfrak{B}[\![B_i]\!](l+1, \pi[\mathtt{P}_{l,1}/\Phi_1, \ldots, \mathtt{P}_{l,n}/\Phi_n]) \quad (1 \le i \le n)
\end{array}$$

  Note, that with the Fixpoint-theorem by Knaster and Tarski (cf. [Wec92]),

$$(\Phi_1, \ldots, \Phi_n) = \bigsqcup \{\mathcal{S}_\Delta^k(\perp_{\underline{\mathrm{C}}}, \ldots, \perp_{\underline{\mathrm{C}}}) \mid\mid k \in \mathbb{N}_0\}$$

  with $\mathcal{S}_\Delta : \underline{\mathrm{C}}^n \dashrightarrow \underline{\mathrm{C}}^n$, defined by

$$(\mathcal{S}_\Delta(\Phi'_1, \ldots, \Phi'_n))_i := \mathfrak{B}[\![B_i]\!](l+1, \pi[\mathtt{P}_{l,1}/\Phi'_1, \ldots, \mathtt{P}_{l,n}/\Phi'_n])$$

and $\perp_{\underline{C}}(\chi')$ undefined for all $\chi' \in \underline{S}$.
The allocation of a variable is performed during the declaration of the variable by adding it to the state. If it was already allocated, its value will be overwritten. In this case, the old value was saved by the auxiliary function 'reset', used in the semantic function for a block. The deallocation is performed during the evaluation of a complete block.

- $\mathfrak{C} : \underline{\texttt{PLP-Cmd}} \times \mathbb{N}_0 \times \underline{U} \times \underline{S} \dashrightarrow \underline{S}$

$$\begin{array}{rcl}
\mathfrak{C}[\![\texttt{X}_{k,o} := E]\!](l,\pi,\chi) & := & \chi[\texttt{X}_{k,o}/\mathfrak{C}[\![E]\!](\chi)],\ \text{if } \chi(\texttt{X}_{k,o}) \text{ defined} \\
\mathfrak{C}[\![\texttt{P}_{k,o}()]\!](l,\pi,\chi) & := & (\pi(\texttt{P}_{k,o}))(\chi) \\
\mathfrak{C}[\![\texttt{if } E \texttt{ then } \Gamma_1 \texttt{ else } \Gamma_2]\!](l,\pi,\chi) & := & \begin{cases} \mathfrak{C}[\![\Gamma_1]\!](l,\pi,\chi), & \text{if } \mathfrak{C}[\![E]\!](\chi) \neq 0 \\ \mathfrak{C}[\![\Gamma_2]\!](l,\pi,\chi), & \text{if } \mathfrak{C}[\![E]\!](\chi) = 0 \end{cases} \\
\mathfrak{C}[\![\underbrace{\texttt{while } E \texttt{ do } \Gamma}_{\Gamma'}]\!](l,\pi,\chi) & := & (\Phi_{\Gamma'})(\chi), \\
& & \text{where } \Phi_{\Gamma'} \text{ is the least fixpoint of} \\
& & (\Phi_{\Gamma'})(\chi') = \begin{cases} \mathfrak{C}[\![\Gamma']\!](l,(\mathfrak{C}[\![\Gamma]\!](l,\pi,\chi'))), & \text{if } \mathfrak{C}[\![E]\!](\chi') \neq 0 \\ \chi', & \text{if } \mathfrak{C}[\![E]\!](\chi') = 0 \end{cases} \\
\mathfrak{C}[\![\texttt{begin } \Gamma_1;\ldots;\Gamma_n \texttt{ end}]\!](l,\pi,\chi) & := & \mathfrak{C}[\![\Gamma_n]\!](l,\ldots(\mathfrak{C}[\![\Gamma_1]\!](l,\pi,\chi))\ldots)
\end{array}$$

- $\mathfrak{B} : \underline{\texttt{PLP-Block}} \times \mathbb{N}_0 \times \underline{U} \times \underline{S} \dashrightarrow \underline{S}$

$$\mathfrak{B}[\![\Delta\Gamma]\!](l,\pi,\chi) \quad := \quad \text{reset}(\chi_l, l, \mathfrak{C}[\![\Gamma]\!](l, \mathfrak{D}[\![\Delta]\!](l,\pi,\chi)))$$

with the auxiliary function reset : $\underline{S} \times \mathbb{N}_0 \times \underline{S} \to \underline{S}$ defined by

$$(\text{reset}(\chi', l', \chi))(\texttt{X}_{l,o}) := \begin{cases} \chi'(\texttt{X}_{l,o}) & \text{if } l = l' \text{ and } \chi'(\texttt{X}_{l,o}) \text{ defined} \\ \text{not defined} & \text{if } l = l' \text{ and } \chi'(\texttt{X}_{l,o}) \text{ not defined} \\ \chi(\texttt{X}_{l,o}) & \text{if } l \neq l' \end{cases}$$

This function restores the values of those variables which may have been changes locally in the block. Variables which are allocated in the block

(i) are deallocated, if they did not exist before the procedure call, and

(ii) are reset to their old values, if they did exist.

- $\mathfrak{M} : \underline{\texttt{PLP-Prog}} \times \mathbb{Z}^\star \dashrightarrow \mathbb{Z}^\star$

$$\begin{array}{rcl}
\mathfrak{M}[\![\texttt{in/out } \texttt{X}_{0,1},\ldots,\texttt{X}_{0,n}; B.]\!](z_1,\ldots,z_n) & := & (\chi(\texttt{X}_{0,1}),\ldots,\chi(\texttt{X}_{0,n})), \\
& & \text{where } \chi = \mathfrak{B}[\![B]\!](1,[\,],[\texttt{X}_{0,1}/z_1,\ldots,\texttt{X}_{0,n}/z_n])
\end{array}$$

We will give an example for the procedure call mechanism; let us take a look at the following program P:

$$
P = \left[\begin{array}{llll}
\texttt{in/out}\ \mathtt{X}_{0,1}; & & & \\
\quad \texttt{var}\ \mathtt{X}_{1,1},\ \mathtt{X}_{1,2}; & & & \\
\quad \texttt{proc}\ \mathtt{P}_{1,1} & & & \\
\qquad \texttt{var}\ \mathtt{X}_{2,1} &]\,\Delta_2 & & \\
\qquad \mathtt{P}_{1,2}(); &]\,\Gamma_2 & B_2 & \Delta_1 \\
\quad \texttt{proc}\ \mathtt{P}_{1,2} & & & \\
\qquad \texttt{var}\ \mathtt{X}_{2,1} &]\,\Delta_3 & & \\
\qquad \mathtt{X}_{2,1} := 1; &]\,\Gamma_3 & B_3 & \\
\ \mathtt{P}_{1,1}(). & & &]\,\Gamma_1
\end{array}\right] B_1
$$

Now we will compute the semantics of P:

$$
\begin{aligned}
\mathfrak{M}[\![P]\!](z) &= (\mathfrak{B}[\![B_1]\!](1,[],[\mathtt{X}_{0,1}/z])(\mathtt{X}_{0,1}))(\mathtt{X}_{0,1}) \\
&= (\mathrm{reset}([],1,\mathfrak{C}[\![\Gamma_1]\!](1,\mathfrak{D}[\![\Delta_1]\!](1,[],[\mathtt{X}_{0,1}/z]))))(\mathtt{X}_{0,1}) \\
&= (\mathrm{reset}([],1,\mathfrak{C}[\![\Gamma_1]\!](1,[\mathtt{P}_{1,1}/\Phi_1,\ \mathtt{P}_{1,2}/\Phi_2],[\mathtt{X}_{0,1}/z,\ \mathtt{X}_{1,1}/0,\ \mathtt{X}_{1,2}/0])))(\mathtt{X}_{0,1}) \\
&\quad\ \text{with } \Phi_i = \mathfrak{B}[\![B_{i+1}]\!](2,[\mathtt{P}_{1,1}/\Phi_1,\ \mathtt{P}_{1,2}/\Phi_2]))(\mathtt{X}_{0,1}) \\
&= (\mathrm{reset}([],1,\Phi_1([\mathtt{X}_{0,1}/z,\ \mathtt{X}_{1,1}/0,\ \mathtt{X}_{1,2}/0])))(\mathtt{X}_{0,1}) \\
&= (\mathrm{reset}([],1, \\
&\qquad \mathfrak{B}[\![B_2]\!](2,[\mathtt{P}_{1,1}/\Phi_1,\ \mathtt{P}_{1,2}/\Phi_2],[\mathtt{X}_{0,1}/z,\ \mathtt{X}_{1,1}/0,\ \mathtt{X}_{1,2}/0])))(\mathtt{X}_{0,1}) \\
&= (\mathrm{reset}([],1,\mathrm{reset}([],2, \\
&\qquad \mathfrak{C}[\![\Gamma_2]\!](2,\mathfrak{D}[\![\Delta_2]\!](2,[\mathtt{P}_{1,1}/\Phi_1,\ \mathtt{P}_{1,2}/\Phi_2],[\mathtt{X}_{0,1}/z,\ \mathtt{X}_{1,1}/0,\ \mathtt{X}_{1,2}/0])))))(\mathtt{X}_{0,1}) \\
&= (\mathrm{reset}([],1,\mathrm{reset}([],2, \\
&\qquad \mathfrak{C}[\![\Gamma_2]\!](2,[\mathtt{P}_{1,1}/\Phi_1,\ \mathtt{P}_{1,2}/\Phi_2],[\mathtt{X}_{0,1}/z,\ \mathtt{X}_{1,1}/0,\ \mathtt{X}_{1,2}/0,\ \mathtt{X}_{2,1}/0]))))(\mathtt{X}_{0,1}) \\
&= (\mathrm{reset}([],1,\mathrm{reset}([],2,\Phi_2([\mathtt{X}_{0,1}/z,\ \mathtt{X}_{1,1}/0,\ \mathtt{X}_{1,2}/0,\ \mathtt{X}_{2,1}/0]))))(\mathtt{X}_{0,1}) \\
&= (\mathrm{reset}([],1,\mathrm{reset}([],2, \\
&\qquad \mathfrak{B}[\![B_3]\!](2,[\mathtt{P}_{1,1}/\Phi_1,\ \mathtt{P}_{1,2}/\Phi_2],[\mathtt{X}_{0,1}/z,\ \mathtt{X}_{1,1}/0,\ \mathtt{X}_{1,2}/0,\ \mathtt{X}_{2,1}/0]))))(\mathtt{X}_{0,1}) \\
&= (\mathrm{reset}([],1,\mathrm{reset}([],2,\mathrm{reset}([\mathtt{X}_{2,1}/0],2, \\
&\qquad \mathfrak{C}[\![\Gamma_3]\!](2,\mathfrak{D}[\![\Delta_3]\!](2,[\mathtt{P}_{1,1}/\Phi_1,\ \mathtt{P}_{1,2}/\Phi_2],[\mathtt{X}_{0,1}/z,\ \mathtt{X}_{1,1}/0,\ \mathtt{X}_{1,2}/0,\ \mathtt{X}_{2,1}/0]))))))(\mathtt{X}_{0,1}) \\
&= (\mathrm{reset}([],1,\mathrm{reset}([],2,\mathrm{reset}([\mathtt{X}_{2,1}/0],2, \\
&\qquad \mathfrak{C}[\![\Gamma_3]\!]([\mathtt{P}_{1,1}/\Phi_1,\ \mathtt{P}_{1,2}/\Phi_2],[\mathtt{X}_{0,1}/z,\ \mathtt{X}_{1,1}/0,\ \mathtt{X}_{1,2}/0,\ \mathtt{X}_{2,1}/0])))))(\mathtt{X}_{0,1}) \\
&= (\mathrm{reset}([],1,\mathrm{reset}([],2,\mathrm{reset}([\mathtt{X}_{2,1}/0],2,[\mathtt{X}_{0,1}/z,\ \mathtt{X}_{1,1}/0,\ \mathtt{X}_{1,2}/0,\ \mathtt{X}_{2,1}/1]))))(\mathtt{X}_{0,1}) \\
&= (\mathrm{reset}([],1,\mathrm{reset}([],2,\mathrm{reset}([\mathtt{X}_{2,1}/0],2,[\mathtt{X}_{0,1}/z,\ \mathtt{X}_{1,1}/0,\ \mathtt{X}_{1,2}/0,\ \mathtt{X}_{2,1}/1]))))(\mathtt{X}_{0,1}) \\
&= (\mathrm{reset}([],1,\mathrm{reset}([],2,[\mathtt{X}_{0,1}/z,\ \mathtt{X}_{1,1}/0,\ \mathtt{X}_{1,2}/0,\ \mathtt{X}_{2,1}/0])))(\mathtt{X}_{0,1}) \\
&= (\mathrm{reset}([],1,[\mathtt{X}_{0,1}/z,\ \mathtt{X}_{1,1}/0,\ \mathtt{X}_{1,2}/0]))(\mathtt{X}_{0,1}) \\
&= ([\mathtt{X}_{0,1}/z])(\mathtt{X}_{0,1}) \\
&= z
\end{aligned}
$$

2 Evolving Algebras

The concept of *evolving algebras* was introduced in [Gur91] as a method for the description of operational semantics. An evolving algebra is the description of a transition system, where the states are algebras over a (first-order) signature. Each state holds the information for a single step of the computation. Recently, evolving algebras were used for a mathematical reconstruction of the WAM ([BR92]). We assume the reader to be familiar with the concept of evolving algebras, so we will not go into detail.

First, we will repeat some standard definitions. Let S be a non-empty set of *sorts* and $\Omega = (\Omega^{(w,s)} \mid\mid w \in S^\star, s \in S)$ a family of sets. Then $\Sigma = (S, \Omega)$ is called a *(many-sorted) signature* and $f \in \Omega^{(w,s)}$ an *operation symbol* with *type* (w, s), *argument sorts* $w = s_1 \dots s_n$ and *target sort* s.

Let $X = (X^s \mid\mid s \in S)$ be a family of sets with *variable symbols* for each sort. The family $\mathrm{T}_\Sigma(X) = (\mathrm{T}^s_\Sigma(X) \mid\mid s \in s)$ of *terms over* Σ *and* X is defined as the smallest family with:

(i) $X^s \subseteq \mathrm{T}^s_\Sigma(X)$

(ii) $f \in \Omega^{(s_1 \dots s_n, s)}, t_1 \in \mathrm{T}^{s_1}_\Sigma(X), \dots t_n \in \mathrm{T}^{s_n}_\Sigma(X) \;\Rightarrow\; f(t_1, \dots, t_n) \in \mathrm{T}^s_\Sigma(X)$

The set $\mathrm{F}_\Sigma(X)$ of *formulas over* Σ *and* X is defined by:

(i) $t_1, t_2 \in \mathrm{T}^s_\Sigma(X) \;\Rightarrow\; t_1 = t_2 \in \mathrm{F}_\Sigma(X)$

(ii) $F, F_1, F_2 \in \mathrm{F}_\Sigma(X) \;\Rightarrow\; (F_1 \vee F_2), (F_1 \wedge F_2), (\neg F) \;\in\; \mathrm{F}_\Sigma(X)$

Let $A = (A^s \mid\mid s \in S)$ be a family of sets. The set $\underline{\mathrm{Ops}}_p^{s_1 \dots s_n, s}(A)$ of *(partial) operations on* A *of type* $(s_1 \dots s_n, s)$ is defined by

$$\underline{\mathrm{Ops}}_p^{s_1 \dots s_n, s}(A) := \{f \mid\mid f : A^{s_1} \times \dots \times A^{s_n} \dashrightarrow A^s\}$$

Let $\alpha = (\alpha^{(w,s)} \mid\mid w \in S^\star, s \in S)$ be a family of functions $\alpha^{(w,s)} : \Omega^{(w,s)} \to \underline{\mathrm{Ops}}_p^{(w,s)}(A)$. Then $\mathfrak{A} = <A, \alpha>$ is called a *(partial)* Σ*-algebra* with *carrier* A.

Let $\beta = (\beta^s \mid\mid s \in S)$ be a family of functions $\beta^s : X^s \to A^s$. We call β a *variable assignment*. This can be extended to $\widehat{\beta} = (\widehat{\beta}^s \mid\mid s \in S)$ with $\widehat{\beta}^s : \mathrm{T}^s_\Sigma(X) \dashrightarrow A^s$ in a unique manner, and we call $\widehat{\beta}(t)$ the *evaluation of* $t \in \mathrm{T}^s_\Sigma(X)$ *under* β. Again, this can be extended to $\widetilde{\beta} : \mathrm{F}_\Sigma(X) \dashrightarrow \{0, 1\}$ and this is called *evaluation of* $F \in \mathrm{F}_\Sigma(X)$ *under* β. The pair $<\mathfrak{A}, \beta>$ *satisfies a formula* F iff $\widetilde{\beta}(F) = 1$. In this case, we say $<\mathfrak{A}, \beta>$ *is a model for* F and we write $<\mathfrak{A}, \beta> \models F$.

There are some differences between the definition of evolving algebras in [BR92] and ours:

- We will only allow *function updates* for the description of the dynamic behavior of evolving algebras. We do not consider *universe extension* or *universe contraction* to be useful, because in the examples of evolving algebras (Modula-2*[GM87], WAM[BR92]) these concepts are used for the design of *memory allocation*. In our opinion this should be part of the abstract machine and not part of the meta-language.

- Due to the fact, that we use a typed signature, we do not need *integrity constraints*, which are just some kind of run-time type check.

A *function update* U has the form

$$U \;=\; f(t_1,\ldots,f_n) := t \quad \text{with } f \in \Omega^{s_1,\ldots,s_n,s},\ t_i \in \mathrm{T}_\Omega^{s_i}(X) \text{ and } t \in \mathrm{T}_\Omega^{s}(X)$$

Let $U_1,\ldots,U_m$ be function updates and $b \in \mathrm{F}_\Omega(X)$ such that each variable in U_j must be contained in b. A *rule* R has the form

$$R = \textbf{if } b \textbf{ then } U_1,\ldots,U_m \textbf{ fi}$$

Let $\mathcal{R}$ be a finite set of rules, $I, F \subset \mathrm{F}_\Omega(X)$ be finite sets and $\varphi = (\varphi^{(w,s)} \mid\mid w \in S^\star, s \in S)$ be a family of functions $\varphi^{(w,s)} : \Omega^{(w,s)} \dashrightarrow \underline{\mathrm{Ops}}_p^{(w,s)}(A)$. This means we have almost an algebra, except that we do not assign an operation to every symbol but only to a selection of symbols. We can identify these symbols by the family of sets $\mathrm{Dom}(\varphi) := (\mathrm{Dom}(\varphi^{(w,s)}) \mid\mid w \in S^\star, s \in S)$. Then we call

$$\mathfrak{A} = <A, \varphi, \mathcal{R}, I, F>$$

a Σ-*evolving algebra* (for short: Σ-EA) with *rules* $\mathcal{R}$, *static operator symbols* $\mathrm{Dom}(\varphi)$, *dynamic operator symbols* $\Omega \setminus \mathrm{Dom}(\varphi)$, *initial conditions*[2] I and *final conditions* F. The set of *states of* $\mathfrak{A}$ is defined by

$$\underline{\mathrm{states}}_\mathfrak{A} := \{\Sigma\text{-Algebras } \mathfrak{S} \mid\mid \mathfrak{S} = <A, \alpha>, \alpha|_{\mathrm{Dom}(\varphi)} = \varphi\}$$

i.e. only the interpretation of the dynamic operation symbols in a state $\mathfrak{S}$ may differ from their interpretation in $\mathfrak{A}$. A state $\mathfrak{S}$ is called *initial (final)* iff $<\mathfrak{S}, \beta> \models I$ $\;(<\mathfrak{S}, \beta> \models F)$ for every variable assignment β.

Now we are able to describe the dynamic behavior of $\mathfrak{A}$. We define the *transition relation* $\vdash_\mathfrak{A} \subseteq \underline{\mathrm{states}}_\mathfrak{A}^2$ by: $\mathfrak{S} = <A, \alpha> \vdash_\mathfrak{A} <A, \alpha'> = \mathfrak{S}'$ iff there exists a variable assignment β with

$$\alpha'(f)(a_1,\ldots,a_n) = \begin{cases} \widehat{\beta}(t) & \text{if } \exists R \in \mathcal{R},\ R = \textbf{if } b \textbf{ then } U_1,\ldots,U_m \textbf{ fi}, \\ & \quad <\mathfrak{S}, \beta> \models b, \\ & \quad \exists 1 \le i \le m : U_i \;=\; f(t_1,\ldots,t_n) := t, \\ & \quad \forall 1 \le j \le n : \widehat{\beta}(t_j) = a_j, \\ \alpha(f)(a_1,\ldots,a_n) & \text{otherwise} \end{cases}$$

for all $f \in \Omega \setminus \mathrm{Dom}(\varphi^{(s_1\ldots s_n,s)})$ and $a_j \in A^{s_j}$ $(\forall 1 \le j \le n)$. Note that all updates in a rule are used simultaneously, i.e. the ordering of the U_i within a rule is of no importance. As usual, $\vdash_\mathfrak{A}^j$ denotes the j-th iteration of $\vdash_\mathfrak{A}$ and $\vdash_\mathfrak{A}^\star$ is the transitive and reflexive closure. A state $\mathfrak{T} \in \underline{\mathrm{states}}_\mathfrak{A}$ is called *reachable* iff there is an initial state $\mathfrak{S} \in \underline{\mathrm{states}}_\mathfrak{A}$ such that $\mathfrak{T} \vdash^\star \mathfrak{S}$.

Note that $\vdash_\mathfrak{A}$ may be non-deterministic (with different variable assignments)[3] and that there may be no transition from a given state, due to inconsistent function updates, i.e. updates, which change the same operation symbol at the same point with different values. The latter may be caused by different rules or different updates within a rule.

[2]Note that there is no connection between the *initial conditions* of an EA and *initial algebras*.

[3]This is different to the behavior in [BR92], where non-determinism is achieved by different rules, applicable at the same time. In this version, all applicable rules are used at the same time, as long as the updates are consistent.

Let $\mathfrak{A}$ be a Σ-EA and $\mathfrak{A}'$ a Σ'-EA. We define $\mathfrak{A}$ to be *sound with regard to* $\mathfrak{A}'$ iff there is an injective function $h : \underline{\text{states}}_{\mathfrak{A}} \dashrightarrow \underline{\text{states}}_{\mathfrak{A}'}$ and $\forall \mathfrak{S}, \mathfrak{T} \in \underline{\text{states}}_{\mathfrak{A}}$:

$$\mathfrak{S} \text{ initial and } \mathfrak{T} \text{ final, } \mathfrak{S} \vdash^{*}_{\mathfrak{A}} \mathfrak{T} \quad \Longrightarrow \quad h(\mathfrak{S}) \vdash^{*}_{\mathfrak{A}'} h(\mathfrak{T}),\ h(\mathfrak{S}) \text{ initial and } h(\mathfrak{T}) \text{ final}$$

Additionally, we define $\mathfrak{A}$ to be *complete with regard to* $\mathfrak{A}'$ iff $\mathfrak{A}'$ is sound with regard to $\mathfrak{A}$, and $\mathfrak{A}$ to be *equivalent to* $\mathfrak{A}'$ iff $\mathfrak{A}$ is sound and complete w.r.t. $\mathfrak{A}'$.

3 The stack machine $\mathfrak{A}_{\text{SM}}$

The machine language `SC` which is used by the stack machine has three different kinds of instructions: data stack instructions, control instructions and procedure stack instructions.

$\underline{\texttt{SC-Inst}}$ $\Gamma ::= \texttt{LIT}\ z \mid \texttt{ADD} \mid$
$\texttt{JMP}\ a \mid \texttt{JMNC}\ a \mid$
$\texttt{LOD}\ (\Delta l, o) \mid \texttt{STO}\ (\Delta l, o) \mid \texttt{CALL}\ (ca, \Delta l, t) \mid \texttt{RET}$
$\underline{\texttt{SC-Prog}}$ $Q ::= \Gamma_1 \ldots \Gamma_n$

As already mentioned, we will describe the stack machine as an EA. First, we have to describe the signature $\Sigma_{\text{SM}} = (S_{\text{SM}}, \Omega_{\text{SM}})$:

- Sorts $S_{\text{SM}} := \{\underline{int}, \underline{intstack}, \underline{scinst}\}$
- Operation symbols $\Omega_{\text{SM}} := \Omega_{\text{SM},s} \mathbin{\dot\cup} \Omega_{\text{SM},d}$, where $\Omega_{\text{SM},s}$ are the static operator symbols and $\Omega_{\text{SM},d}$ are the dynamic operator symbols.

$$\begin{array}{lcl}
\Omega_{\text{SM},s} & := & \{\ \mathit{CODE}\ (\underline{int}, \underline{scinst}),\ \mathit{arity}\ (\underline{int}),\\
& & \quad \mathit{0}\ (\underline{int}),\ \mathit{1}\ (\underline{int}),\ \mathit{2}\ (\underline{int}),\ \mathit{3}\ (\underline{int}),\ +\ (\underline{int}\cdot\underline{int}, \underline{int}),\\
& & \quad \texttt{LIT}\ (\underline{int}, \underline{scinst}),\ \texttt{ADD}\ (\underline{scinst}),\\
& & \quad \texttt{JMP}\ (\underline{int}, \underline{scinst}),\ \texttt{JMNC}\ (\underline{int}, \underline{scinst}),\\
& & \quad \texttt{LOD}\ (\underline{int}\cdot\underline{int}, \underline{scinst}),\ \texttt{STO}\ (\underline{int}\cdot\underline{int}, \underline{scinst}),\\
& & \quad \texttt{CALL}\ (\underline{int}\cdot\underline{int}\cdot\underline{int}, \underline{scinst}),\ \texttt{RET}\ (\underline{scinst})\\
& & \quad \mathit{push}\ (\underline{intstack}\cdot\underline{int}, \underline{intstack}),\ \mathit{pushn0}\ (\underline{intstack}\cdot\underline{int}, \underline{intstack}),\\
& & \quad \mathit{pop}\ (\underline{intstack}, \underline{intstack}),\ \mathit{popn}\ (\underline{intstack}\cdot\underline{int}, \underline{intstack}),\ \mathit{top}\ (\underline{intstack}, \underline{int}),\\
& & \quad \mathit{select}\ (\underline{intstack}\cdot\underline{int}, \underline{int}),\ \mathit{modify}\ (\underline{intstack}\cdot\underline{int}\cdot\underline{int}, \underline{intstack}),\\
& & \quad \mathit{height}\ (\underline{intstack}, \underline{int}), \mathit{base}\ (\underline{intstack}\cdot\underline{int}, \underline{int})\}\\
\Omega_{\text{SM},d} & := & \{\ \mathit{PC}\ (\underline{int}),\ \mathit{DS}\ (\underline{intstack}),\ \mathit{PS}\ (\underline{intstack})\}
\end{array}$$

During the execution, *CODE* holds the *stack code* and *PC* is the *program counter*. The *data stack* is represented by *DS*, and *PS* is the *procedure stack*. The operator symbols `LIT`, `ADD`, …,`RET` are used to decode the next command during execution. The actual changes in the state are caused by *push*,*pop*,…,*base*. The symbol *arity* is for technical reasons only; it holds the number of `in/out`-variables of the source program and it is used only to ensure that the initial states are valid.

Now, we have to give a meaning to the static symbols. Not surprisingly, the sorts are interpreted in the following way:

$$A_{\text{SM}} = (A^{\underline{int}}, A^{\underline{intstack}}, A^{\underline{scinst}})$$

with $A^{\underline{int}} := \mathbb{Z}$, $A^{\underline{intstack}} := \mathbb{Z}^\star$ and $A^{\underline{scinst}} :=$ $\underline{\texttt{SC-Inst}}$.

The static operation symbols get their meaning by φ_{SM}:

$$\begin{aligned}
\varphi_{\text{SM}}(\mathit{0}) &:= 0 \\
\varphi_{\text{SM}}(\mathit{1}) &:= 1 \\
\varphi_{\text{SM}}(\mathit{2}) &:= 2 \\
\varphi_{\text{SM}}(\mathit{3}) &:= 3 \\
\varphi_{\text{SM}}(+)(z_1, z_2) &:= z_1 + z_2 \\
\varphi_{\text{SM}}(\texttt{LIT})(z) &:= \texttt{LIT}\ z \\
\varphi_{\text{SM}}(\texttt{ADD}) &:= \texttt{ADD} \\
\varphi_{\text{SM}}(\texttt{JMP})(a) &:= \texttt{JMP}\ a \\
\varphi_{\text{SM}}(\texttt{JMNC})(a) &:= \texttt{JMNC}\ a \\
\varphi_{\text{SM}}(\texttt{LOD})(\Delta l, o) &:= \texttt{LOD}\ (\Delta l, o) \\
\varphi_{\text{SM}}(\texttt{STO})(\Delta l, o) &:= \texttt{STO}\ (\Delta l, o) \\
\varphi_{\text{SM}}(\texttt{CALL})(ca, \Delta l, t) &:= \texttt{CALL}\ (ca, \Delta l, t) \\
\varphi_{\text{SM}}(\texttt{RET}) &:= \texttt{RET} \\
\varphi_{\text{SM}}(\mathit{push})(z_1 : \cdots : z_n, z) &:= z : z_1 : \cdots : z_n \\
\varphi_{\text{SM}}(\mathit{pushn0})(z_1 : \cdots : z_n, z) &:= \underbrace{0 : \ldots : 0}_{z} : z_1 : \cdots : z_n \\
\varphi_{\text{SM}}(\mathit{pop})(z_1 : \cdots : z_n) &:= z_2 : \cdots : z_n \quad (1 \le n) \\
\varphi_{\text{SM}}(\mathit{popn})(z_1 : \cdots : z_n, m) &:= z_{m+1} : \cdots : z_n \quad (1 \le m \le n) \\
\varphi_{\text{SM}}(\mathit{top})(z_1 : \cdots : z_n) &:= z_1 \\
\varphi_{\text{SM}}(\mathit{select})(z_1 : \cdots : z_n, i) &:= z_i \quad (1 \le i \le n) \\
\varphi_{\text{SM}}(\mathit{modify})(z_1 : \cdots : z_n, i, z) &:= z_1 : \cdots : z_{i-1} : z : z_{i+1} : \cdots : z_n \quad (1 \le i \le n) \\
\varphi_{\text{SM}}(\mathit{height})(z_1 : \cdots : z_n) &:= n \\
\varphi_{\text{SM}}(\mathit{base})(z_1 : \cdots : z_n, \Delta l) &:= \text{base}(z_1 : \cdots : z_n, \Delta l)
\end{aligned}$$

with the auxiliary function $\text{base} : \mathbb{Z}^\star \times \mathbb{Z} \dashrightarrow \mathbb{Z}$ defined by

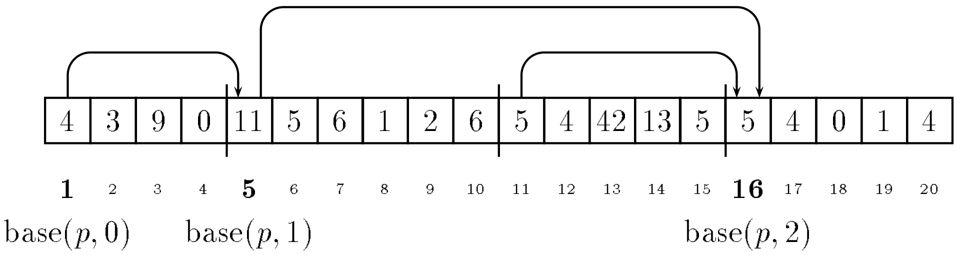
$$\begin{aligned}
\text{base}(p, 0) &:= 1 \\
\text{base}(p, \Delta l + 1) &:= \text{base}(p, \Delta l) + p.(\text{base}(p, \Delta l))
\end{aligned}$$

Of course, the definitions of *CODE* and *arity* are missing; they will be inserted by the translation mapping.

The 'base'-function is the central point of the static-link technique; it allows indirect address calculation on the procedure stack, using certain stack entries as relative addresses (static links). For example, with $p = 4 : 3 : 9 : 0 : 11 : 5 : 6 : 1 : 2 : 6 : 5 : 3 : 42 : 13 : 5 : 4 : 0 : 1 : 4$ we have

$$\begin{aligned}
\mathrm{base}(p, 0) &= 1 \\
\mathrm{base}(p, 1) &= 1 + 4 = 5 \\
\mathrm{base}(p, 2) &= 5 + 11 = 16
\end{aligned}$$

and this can be visualized as:



The stack p consists of four *activation records*, each representing a procedure call, with entries for the local variables and for the return address. Additionally, there are two administrative entries, the static link and the dynamic link. The dynamic link is used to determine the length of an activation record, and the static link is used by the 'base'-function to access inner activation record. In the example, the third activation record from the top of the stack is hidden by the static link of the second activation record, i.e. only the variables in the first, second and fourth record can be accessed.

We now introduce the rules $\mathcal{R}_{\text{SM}}$. Although there are free variables (the parameters of the machine instructions) in the conditions of the rules, the stack machine is deterministic.

$\mathcal{R}_{\text{SM}} = \{$**if** $\mathit{CODE}(\mathit{PC}) = \texttt{LIT}(z)$ **then**
    $\mathit{PC} := \mathit{PC} + 1$
    $\mathit{DS} := \mathit{push}(\mathit{DS}, z)$
  fi,
  if $\mathit{CODE}(\mathit{PC}) = \texttt{ADD}$ **then**
    $\mathit{PC} := \mathit{PC} + 1$
    $\mathit{DS} := \mathit{push}(\mathit{pop}(\mathit{pop}(\mathit{DS})), \mathit{top}(\mathit{DS}) + \mathit{top}(\mathit{pop}(\mathit{DS})))$
  fi,
  if $\mathit{CODE}(\mathit{PC}) = \texttt{JMP}(a)$ **then**
    $\mathit{PC} := a$
  fi,
  if $\mathit{CODE}(\mathit{PC}) = \texttt{JMNC}(a) \ \wedge\ \mathit{top}(\mathit{DS}) = 0$ **then**
    $\mathit{PC} := a$
    $\mathit{DS} := \mathit{pop}(\mathit{DS})$
  fi,

$$
\begin{array}{l}
\textbf{if } \mathit{CODE}(\mathit{PC}) = \texttt{JMNC}(a) \ \wedge \ \mathit{top}(\mathit{DS}) \neq 0 \textbf{ then} \\
\quad \mathit{DS} := \mathit{pop}(\mathit{DS}) \\
\textbf{fi}, \\
\textbf{if } \mathit{CODE}(\mathit{PC}) = \texttt{LOD}(\Delta l, o) \textbf{ then} \\
\quad \mathit{PC} := \mathit{PC} + 1 \\
\quad \mathit{DS} := \mathit{push}(\mathit{DS}, \mathit{select}(\mathit{PS}, \mathit{base}(\mathit{PS}, \Delta l) + o + 2)) \\
\textbf{fi}, \\
\textbf{if } \mathit{CODE}(\mathit{PC}) = \texttt{STO}(\Delta l, o) \textbf{ then} \\
\quad \mathit{PC} := \mathit{PC} + 1 \\
\quad \mathit{PS} := \mathit{modify}(\mathit{PS}, \mathit{base}(\mathit{PS}, \Delta l) + o + 2, \mathit{top}(\mathit{DS})) \\
\quad \mathit{DS} := \mathit{pop}(\mathit{DS}) \\
\textbf{fi}, \\
\textbf{if } \mathit{CODE}(\mathit{PC}) = \texttt{CALL}(ca, \Delta l, t) \textbf{ then} \\
\quad \mathit{PC} := ca \\
\quad \mathit{PS} := \mathit{push}(\mathit{push}(\mathit{push}(\mathit{pushn0}(\mathit{PS}, t), \mathit{PC} + 1), t + 2), \mathit{base}(\mathit{PS}, \Delta l) + t + 2) \\
\textbf{fi}, \\
\textbf{if } \mathit{CODE}(\mathit{PC}) = \texttt{RET} \textbf{ then} \\
\quad \mathit{PC} := \mathit{top}(\mathit{pop}(\mathit{pop}(\mathit{PS}))) \\
\quad \mathit{PS} := \mathit{popn}(\mathit{PS}, \mathit{top}(\mathit{pop}(\mathit{PS})) + 1) \\
\textbf{fi}
\end{array}
$$

The rules for `LIT`, `ADD`, `JMP` and `JMNC` only access the data stack and the program counter in the expected way. Access to the procedure stack in done in the rule for

- `LOD`, which puts a certain element of the procedure stack onto the data stack. As we will see later, this element corresponds to a variable in the source language.
- `STO`, which replaces the same element by the topmost data stack cell.
- `CALL`, which creates an activation record of the form

  $$sl : dl : ra : \underbrace{0 : \ldots : 0}_{t}$$

  where sl is the static link, dl is the dynamic link giving the length of the activation record, ra is the return address and a block of t zeros, which represents the local variables of the called procedure.
- `RET`, which removes the topmost activation record and jumps to the appropriate return address.

In order to complete the definition of the stack machine, we have to define the initial and final conditions:

$$
\begin{array}{rcl}
I_{\mathrm{SM}} & := & \{\mathit{PC} = 1,\ \mathit{height}(\mathit{PS}) = \mathit{arity} + 3,\ \mathit{top}(\mathit{PS}) = \mathit{arity} + 3,\ \mathit{top}(\mathit{pop}(\mathit{PS})) = \mathit{arity} + 2\} \\
F_{\mathrm{SM}} & := & \{\mathit{PC} = 0\}
\end{array}
$$

Program execution starts at the first command in the program and terminates as soon as the program counter is set to *0*. Additionally, on program start, the procedure stack PS must be a single activation record with the correct number of variables.

4 The translation mapping

The translation mapping inserts the components $CODE$ and *arity* of the stack machine. It constructs a `SC`-program from a given `PLP`-program, i.e. it is a `PLP`-compiler. We will present a slightly modified version of the translation mapping in [Ind88]:

(i) we do not need to record the current level in the symbol table, because the identifiers themselves contain this information.

(ii) we do not use symbolic addresses for jumps, because we want to avoid the second compilation phase. This is achieved by the auxiliary function 'len', which precomputes the length of the compiled code of a `PLP`-construct.

len : $\underline{\texttt{PLP-Exp}} \cup \underline{\texttt{PLP-Decl}} \cup \underline{\texttt{PLP-Cmd}} \cup \underline{\texttt{PLP-Block}} \rightarrow \mathbb{N}_0$

$$\begin{array}{rcl}
\mathrm{len}[\![z]\!] & := & 1 \\
\mathrm{len}[\![\mathtt{X}_{l,o}]\!] & := & 1 \\
\mathrm{len}[\![E_1 + E_2]\!] & := & \mathrm{len}[\![E_1]\!] + \mathrm{len}[\![E_2]\!] + 1 \\
\mathrm{len}[\![\Delta_v \Delta_p]\!] & := & \mathrm{len}[\![\Delta_p]\!] \\
\mathrm{len}[\![\varepsilon]\!] & := & 0 \\
\mathrm{len}[\![\mathtt{proc}\ \mathtt{P}_{l,1}; B_1; \ldots; \mathtt{P}_{l,n}; B_n;]\!] & := & \sum_{j=1}^{n} \mathrm{len}[\![B_j]\!] \\
\mathrm{len}[\![\mathtt{X}_{l,o} := E]\!] & := & \mathrm{len}[\![E]\!] + 1 \\
\mathrm{len}[\![\mathtt{P}_{l,o}()]\!] & := & 1 \\
\mathrm{len}[\![\mathtt{if}\ E\ \mathtt{then}\ \Gamma_1\ \mathtt{else}\ \Gamma_2]\!] & := & \mathrm{len}[\![E]\!] + \mathrm{len}[\![\Gamma_1]\!] + \mathrm{len}[\![\Gamma_2]\!] + 2 \\
\mathrm{len}[\![\mathtt{while}\ E\ \mathtt{do}\ \Gamma]\!] & := & \mathrm{len}[\![E]\!] + \mathrm{len}[\![\Gamma]\!] + 2 \\
\mathrm{len}[\![\mathtt{begin}\ \Gamma_1; \ldots; \Gamma_n\ \mathtt{end}]\!] & := & \sum_{j=1}^{n} \mathrm{len}[\![\Gamma_j]\!] \\
\mathrm{len}[\![\Delta\Gamma]\!] & := & \mathrm{len}[\![\Delta]\!] + \mathrm{len}[\![\Gamma]\!] + 1
\end{array}$$

Another auxiliary function computes the number of local variables defined in a block.
size : $\underline{\texttt{PLP-Block}} \rightarrow \mathbb{N}_0$

$$\begin{array}{rcl}
\mathrm{size}[\![\Delta_p \Gamma]\!] & := & 0 \\
\mathrm{size}[\![\mathtt{var}\ \mathtt{X}_{l,1}, \ldots, \mathtt{X}_{l,n} \Delta_p \Gamma]\!] & := & n
\end{array}$$

In the denotational semantics the environment holds information about identifiers. The translation mapping uses a *symbol table* in a similar way. The set of symbol tables is defined by:

$$\underline{\mathrm{Tab}} := \{st \mid\mid st : (\underline{\texttt{VIde}} \,\dot\cup\, \underline{\texttt{PIde}}) \dashrightarrow (\{\mathtt{var}\} \,\dot\cup\, (\{\mathtt{proc}\} \times \mathbb{N}_0 \times \mathbb{N}_0))\}$$

We associate a dummy entry with a variable identifier and the address of the first command of the code and the number of local variables with a procedure identifier. The dummy entry for the variables is used to check whether the variable is defined.

The function 'upd' updates the symbol table with regard to a declaration. An additional argument contains the next available code address.

upd : $\underline{\texttt{PLP-Decl}} \times \underline{\text{Tab}} \times \mathbb{N}_0 \dashrightarrow \underline{\text{Tab}}$

$$
\begin{array}{rcl}
\text{upd}[\![\Delta_v \Delta_p]\!](st, a) & := & \text{upd}[\![\Delta_p]\!](\text{upd}[\![\Delta_v]\!](st, a), a) \\
\text{upd}[\![\varepsilon]\!](st, a) & := & st \\
\text{upd}[\![\texttt{var } \texttt{X}_{l,1}, \ldots, \texttt{X}_{l,n}]\!](st, a) & := & st[\texttt{X}_{l,1}/(\texttt{var}), \ldots, \texttt{X}_{l,n}/(\texttt{var})] \\
\text{upd}[\![\texttt{proc } \texttt{P}_{l,1}; B_1; \ldots; \texttt{P}_{l,n}; B_n]\!](st, a) & := & st[\texttt{P}_{l,1}/(\texttt{proc}, a, \text{size}[\![B_1]\!]), \\
 & & \quad \texttt{P}_{l,2}/(\texttt{proc}, a + \text{len}[\![B_1]\!], \text{size}[\![B_2]\!]) \\
 & & \quad \vdots \\
 & & \quad \texttt{P}_{l,n}/(\texttt{proc}, a + \sum_{k=1}^{n-1} \text{len}[\![B_k]\!], \text{size}[\![B_n]\!])]
\end{array}
$$

Now we are able to define the translation mappings. The main translation mapping is 'trans', which takes an PLP-program as its argument and creates the corresponding SC-program. It uses several auxiliary mappings, one for each possible component of a PLP-program. These mappings all have a similar type; their first argument is the component they translate; the second and third are the current symbol table and the current nesting level. All mappings except 'et' (which cannot create SC-commands for control flow) have a fourth argument, which is the next free code address.

trans : $\underline{\texttt{PLP-Prog}} \dashrightarrow \underline{\texttt{SC-Prog}}$

$$
\text{trans}[\![\texttt{in/out } \texttt{X}_{0,1}, \ldots, \texttt{X}_{0,n}; B.]\!] := \left[\begin{array}{l}
\texttt{CALL } (3, 0, \text{size}[\![B]\!]) \\
\texttt{JMP 0} \\
\text{bt}[\![B]\!]([(\texttt{X}_{0,1}/(\texttt{var}), \ldots, (\texttt{X}_{0,n}/(\texttt{var})], 3, 1)
\end{array}\right.
$$

bt : $\underline{\texttt{PLP-Block}} \times \underline{\text{Tab}} \times \mathbb{N}_0 \times \mathbb{N}_0 \dashrightarrow \underline{\texttt{SC-Prog}}$

$$
\text{bt}[\![\Delta\Gamma]\!](st, l, a) := \left[\begin{array}{l}
\text{ct}[\![\Gamma]\!](\text{upd}[\![\Delta]\!](st, a + \text{len}[\![\Gamma]\!] + 1), l, a) \\
\texttt{RET} \\
\text{dt}[\![\Delta]\!](\text{upd}[\![\Delta]\!](st, a + \text{len}[\![\Gamma]\!] + 1), l, a + \text{len}[\![\Gamma]\!] + 1)
\end{array}\right.
$$

dt : $\underline{\texttt{PLP-Decl}} \times \underline{\text{Tab}} \times \mathbb{N}_0 \times \mathbb{N}_0 \dashrightarrow \underline{\texttt{SC-Prog}}$

$$
\begin{array}{rcl}
\text{dt}[\![\Delta_v \Delta_p]\!](st, l, a) & := & \text{dt}[\![\Delta_p]\!](st, l, a) \\
\text{dt}[\![\varepsilon]\!](st, l, a) & := & \varepsilon \\
\text{dt}[\![\texttt{proc } \texttt{P}_{l,1}; B_1; \ldots; \texttt{P}_{l,n}; B_n]\!](st, l, a) & := & \left[\begin{array}{l}
\text{bt}[\![B_1]\!](st, l+1, a) \\
\text{bt}[\![B_2]\!](st, l+1, a + \text{len}[\![B_1]\!]) \\
\vdots \\
\text{bt}[\![B_n]\!](st, l+1, a + \sum_{k=1}^{n-1} \text{len}[\![B_k]\!])
\end{array}\right.
\end{array}
$$

$\mathrm{ct} : \underline{\texttt{PLP-Cmd}} \times \underline{\mathrm{Tab}} \times \mathbb{N}_0 \times \mathbb{N}_0 \dashrightarrow \underline{\texttt{SC-Prog}}$

$$\begin{array}{rcl}
\mathrm{ct}[\![\mathtt{X}_{dl,o} := E]\!](st,l,a) & := & \left[\begin{array}{l}\mathrm{et}[\![E]\!](st,l)\\ \mathtt{STO}\ (l-dl,o)\end{array}\right. \quad \text{if } st(\mathtt{X}_{dl,o}) = (\mathtt{var})\\
\mathrm{ct}[\![\mathtt{P}_{dl,o}]\!](st,l,a) & := & \mathtt{CALL}\ (ca, l-dl, t) \quad \text{if } st(\mathtt{P}_{dl,o}) = (\mathtt{proc}, ca, t)\\
\mathrm{ct}[\![\mathtt{if}\ E\ \mathtt{then}\ \Gamma_1\ \mathtt{else}\ \Gamma_2]\!](st,l,a) & := & \left[\begin{array}{l}\mathrm{et}[\![E]\!](st,l)\\ \mathtt{JMNC}\ a+\mathrm{len}[\![E]\!]+\mathrm{len}[\![\Gamma_1]\!]+2\\ \mathrm{ct}[\![\Gamma_1]\!](st,l,a+\mathrm{len}[\![E]\!]+1)\\ \mathtt{JMP}\ a+\mathrm{len}[\![E]\!]+\mathrm{len}[\![\Gamma_1]\!]+\mathrm{len}[\![\Gamma_2]\!]+2\\ \mathrm{ct}[\![\Gamma_2]\!](st,l,a+\mathrm{len}[\![E]\!]+1)\end{array}\right.\\
\mathrm{ct}[\![\mathtt{while}\ E\ \mathtt{do}\ \Gamma]\!](st,l,a) & := & \left[\begin{array}{l}\mathrm{et}[\![E]\!](st,l)\\ \mathtt{JMNC}\ (a+\mathrm{len}[\![E]\!]+\mathrm{len}[\![\Gamma]\!]+2)\\ \mathrm{ct}[\![\Gamma]\!](st,l,a+\mathrm{len}[\![E]\!]+1)\\ \mathtt{JMP}\ a\end{array}\right.\\
\mathrm{ct}[\![\mathtt{begin}\ \Gamma_1;\ldots;\Gamma_n\ \mathtt{end}]\!](st,l,a) & := & \left[\begin{array}{l}\mathrm{ct}[\![\Gamma_1]\!](st,l,a)\\ \mathrm{ct}[\![\Gamma_2]\!](st,l,a+\mathrm{len}[\![\Gamma_1]\!])\\ \vdots\\ \mathrm{ct}[\![\Gamma_n]\!](st,l,a+\sum_{k=1}^{n-1}\mathrm{len}[\![\Gamma_k]\!])\end{array}\right.
\end{array}$$

$\mathrm{et} : \underline{\texttt{PLP-Exp}} \times \underline{\mathrm{Tab}} \times \mathbb{N}_0 \dashrightarrow \underline{\texttt{SC-Prog}}$

$$\begin{array}{rcl}
\mathrm{et}[\![z]\!](st,l) & := & \mathtt{LIT}\ z\\
\mathrm{et}[\![\mathtt{X}_{dl,o}]\!](st,l) & := & \mathtt{LOD}\ (l-dl,o) \quad \text{if } st(\mathtt{X}_{dl,o}) = (\mathtt{var})\\
\mathrm{et}[\![E_1+E_2]\!](st,l) & := & \left[\begin{array}{l}\mathrm{et}[\![E_1]\!](st,l)\\ \mathrm{et}[\![E_2]\!](st,l)\\ \mathtt{ADD}\end{array}\right.
\end{array}$$

This translation mapping creates the same code as the one in [Ind88].

One can easily verify that 'len' computes the correct result, i.e. for each $B \in \underline{\texttt{PLP-Block}}$ (and similar for $\Delta \in \underline{\texttt{PLP-Decl}}$ etc.), $st \in \underline{\mathrm{Tab}}$ and $a, l \in \mathbb{N}_0$ holds:

$$\Gamma_1,\ldots,\Gamma_n = \mathrm{bt}[\![B]\!](st,l,a) \quad \Longrightarrow \quad \mathrm{len}[\![B]\!] = n$$

Now we are in the position to complete the definition of the stack machine. If we have a program $P = \mathtt{in/out}\ \mathtt{X}_{0,1}\ldots,\mathtt{X}_{0,n}; B. \in \underline{\texttt{PLP-Prog}}$ and $\Gamma_1\ldots,\Gamma_r = \mathrm{trans}[\![P]\!]$, then we call the Σ_{SM}-EA

$$\mathfrak{A}_{\mathrm{SM}}(P) := <A_{\mathrm{SM}}, \varphi_{\mathrm{SM},P}, \mathcal{R}_{\mathrm{SM}}, I_{\mathrm{SM}}, F_{\mathrm{SM}}>$$

with $\varphi_{\mathrm{SM},P} := \varphi_{\mathrm{SM}} \cup \{CODE : i \mapsto \Gamma_i\ (1 \le i \le r)\} \cup \{arity \mapsto n\}$ the *stack machine for P*.

5 Everything you always wanted to know about the 'base'-function, but were afraid to ask

In the definition of $\mathfrak{A}_{\mathrm{SM}}$ the procedure stack PS is of type *intstack*, i.e. its value can be an arbitrary word over the integers. Of course, not all possible words can be values of PS during a computation starting in an initial state. As already mentioned, the stack possesses a certain structure during program execution. It is vital for the correctness of the machine, that this structure is preserved, because only this structure guarantees that the 'base'-function works properly and that it accesses variables in a correct way. It is the aim of this section to prove that this structure is respected during program execution.

Definition 1 (admissible program stack) A word $p \in \mathbb{Z}^\star$ is called an **admissible program stack** with $h \in \mathbb{N}$ **blocks** iff

(i) There exists $b_1, \ldots, b_h \in \mathbb{Z}^\star$ blocks with:

$$p = b_h b_{h-1} \cdots b_1$$

(ii) $\forall 1 \leq i \leq h\ \exists sl_i \in \mathbb{N}, dl_i \in \mathbb{N}, sl_i \geq 3, dl_i \geq 2, ra \in \mathbb{N}_0, lv_{i,1}, \ldots, lv_{i,dl_i-2} \in \mathbb{Z}$:

$$b_i = sl_i : dl_i : ra_i : lv_{i,1} : \ldots : lv_{i,dl_i-2}$$

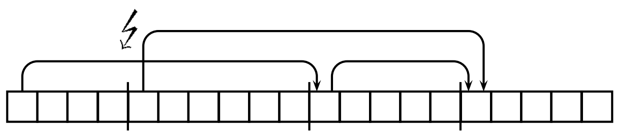
(iii) For every block $1 \leq i \leq h$ there is a block $1 \leq j \leq i$ with:

$$sl_i = \sum_{k=j}^{i} |b_k| = \sum_{k=j}^{i} (dl_k + 1)$$

(iv) If two blocks j and i are connected by (iii) we have for each block $j \leq i' < i$ between them

$$sl_{i'} = \sum_{k'=j'}^{i'} |b_{k'}| \quad \Longrightarrow \quad j \geq j' \qquad \triangleleft$$

The first condition ensures that the stack consists of blocks. The structure of each block is fixed by condition (ii), which states that each block has to have at least three elements: a static link, a dynamic link, which determines the length of the block, a return address and (depending on the number of local variables of the last procedure call) entries for variables. Condition (iii) ensures that each dynamic link is a pointer to another dynamic link in a lower block. As a consequence of the fourth and last condition, these pointer may not cross, i.e. a situation like the following is not allowed for an admissible procedure stack:



As already mentioned, it is the aim of this section to prove that the execution of a compiled PLP-program respects this structure, i.e. that the value of PS is admissible. Of course, it is easy to see that the commands CALL and RET transform an admissible stack into a new admissible stack by appending (resp. deleting) an activation block. A problem arises with the STO-command. It replaces a single entry 'somewhere' in the procedure stack. It is not trivial to show that these operations do not affect a static or dynamic link, which would destroy the structure.

First of all, we will examine the relation between the 'base'-function and admissible procedure stacks.

Corollary 1 Let $p \in \mathbb{Z}^\star$ be admissible with h blocks and $sl_h = \sum\limits_{k=m}^{h} |b_k|$ with $1 \leq m \leq h$. Then:

$$\mathrm{base}(p,1) = 1 + \sum_{k=m}^{h} |b_k| \qquad \diamondsuit$$

Proof $\mathrm{base}(p,1) = 1 + p.1 = 1 + sl_h = 1 + \sum\limits_{k=m}^{h} |b_k|$ q.e.d.

Lemma 1 Let $p \in \mathbb{Z}^\star$ be admissible with h blocks and let $p' = p.(\mathrm{base}(p,1)) : \ldots : p.|p|$. Then:

(i) if $\mathrm{base}(p,1) < |p|$ then p' is admissible and the number of blocks of p' is $m-1$, if $p.1 = \sum\limits_{k=m}^{h} |b_k|$.

(ii) if $p' \neq \varepsilon$ then $\mathrm{base}(p,2) = \mathrm{base}(p',1) + p.1$. $\diamondsuit$

Proof With Corollary 1 we have: $\mathrm{base}(p,1) = 1 + p.1 = \sum\limits_{k=m}^{h} |b_k| \geq 0$.
On the other hand, we have:

$$\begin{aligned}
|p'| &= |p| - (\mathrm{base}(p,1) - 1) \\
&= |p| - (1 + \sum_{k=m}^{h} |b_k| - 1) \\
&= |p| - \sum_{k=m}^{h} |b_k|
\end{aligned}$$

This means, that $p' = b_{m-1} : \ldots : b_1$ and this is admissible iff $p' \neq \varepsilon$. To ensure that, we have to show that $m > 1$:

$$\begin{aligned}
\mathrm{base}(p,1) < |p| &\iff |p| - \mathrm{base}(p,1) > 0 \\
&\iff \sum_{k=1}^{h} |b_k| - \sum_{k=m}^{h} |b_k| - 1 > 0 \\
&\iff m > 1
\end{aligned}$$

This completes the proof for (i).
Finally, we have to show (ii). Let $p' \neq \varepsilon$, which is equivalent to $m > 2$:

$$\begin{aligned}
\mathrm{base}(p,2) &= \mathrm{base}(p,1) + p.(\mathrm{base}(p,1)) \\
&= 1 + p.1 + p'.1 \\
&= \mathrm{base}(p',1) + p.1
\end{aligned}$$

q.e.d.

Corollary 2 If $p \in \mathbb{Z}^\star$ is admissible with h blocks then there exists a maximal $\mathrm{l}(p) \geq 0$ with

$$p.\mathrm{base}(p, \Delta l) \text{ defined (i.e. } \mathrm{base}(p, \Delta l) \leq |p|) \quad \forall 0 \leq \Delta l \leq \mathrm{l}(p)$$

If $p' = p.(\mathrm{base}(p,1)) : \ldots : p.|p|$ and $p' \neq \varepsilon$ then for all $2 \leq \Delta l \leq \mathrm{l}(p)$ holds:

$$\mathrm{base}(p, \Delta l) = \mathrm{base}(p', \Delta l - 1) + p.1$$

We call $\mathrm{l}(p)$ the **maximal chain length of $\boldsymbol{p}$**. $\diamondsuit$

Proof: Successive application of Lemma 1.

Corollary 3 (Generalization of Corollary 1) Let $p \in \mathbb{Z}^\star$ be admissible with h blocks. For all $1 \leq \Delta l \leq \mathrm{l}(p)$ exists $1 \leq m \leq h$ with:

$$\mathrm{base}(p, \Delta l) = 1 + \sum_{k=m}^{h} |b_k| \qquad \diamondsuit$$

Proof Induction on l:
$\Delta l = 1$: Corollary 1
$\Delta l \to \Delta l + 1$: let $\Delta l + 1 \leq \mathrm{l}(p)$ and $p' = p.(\mathrm{base}(p,1)) : \ldots : p.|p|$. With Lemma 1 we know that p' is admissible. Let h' be the number of blocks of p'. With the induction hypothesis we get a $m \in \{1, \ldots, h'\}$ such that:

$$\mathrm{base}(p', \Delta l) = 1 + \sum_{k=m}^{h'} |b_k|$$

From this, we can deduce with Corollary 2:

$$\begin{aligned}
\mathrm{base}(p, \Delta l + 1) &= p.1 + \mathrm{base}(p', \Delta l) \\
&= p.1 + 1 + \sum_{k=m}^{h'} |b_k| \\
&= \sum_{k=h'+1}^{h} |b_k| + 1 + \sum_{k=m}^{h'} |b_k| \\
&= 1 + \sum_{k=m}^{h} |b_k|
\end{aligned}$$

q.e.d.

Now we can turn to the relationship between admissible procedure stacks and the execution of compiled `PLP`-programs. As already mentioned, the only problem is the `STO` command.

Lemma 2 Let $P \in$ <u>`PLP-Prog`</u>, $\mathfrak{A}_{\mathrm{SM}}(P)$ the stack machine for P, $\mathfrak{S} = <A, \alpha> \in \underline{\mathrm{states}}_{\mathfrak{A}_{\mathrm{SM}}(P)}$ a state and let $\alpha(PS)$ be admissible with h blocks.

(i) If $\alpha(CS)(\alpha(PC)) = \texttt{CALL}\ (ca, \Delta l, t)$ with $\Delta l \leq \mathrm{l}(p)$ then:

$$\mathfrak{S} \vdash_{\mathrm{SM}} \mathfrak{S}' = <A, \alpha'>$$

is defined and $\alpha'(PS)$ is admissible with $h+1$ blocks and

$$\mathrm{l}(\alpha'(PS)) = \mathrm{l}(\alpha(PS)) - \Delta l + 1$$

(ii) If $\alpha(CS)(\alpha(PC)) = \texttt{RET}$ with $h \geq 1$ then:

$$\mathfrak{S} \vdash_{\mathrm{SM}} \mathfrak{S}' = <A, \alpha'>$$

is defined and $\alpha'(PS)$ is admissible with $h-1$ blocks. ♢

Proof

(i) The new value of PS is $\alpha'(PS) = b_{h+1} : \alpha(PS)$ with

- $b_{h+1} := dl_{h+1} : sl_{h+1} : ra_{h+1} : lv_{h+1,1} : \ldots : lv_{h+1,t}$
- $dl_{h+1} := \mathrm{base}(\alpha(PS), \Delta l) + 2 + t$
- $sl_{h+1} := t + 2$
- $ra_{h+1} := \alpha(PC) + 1$
- $lv_{h+1,1} := \ldots := lv_{h+1,t} := 0$

Now we have to prove condition (iii) and (iv) of Definition 1 for b_{h+1}:

- $\exists 1 \leq j \leq h+1$ with $sl_{h+1} = \sum_{k=j}^{h+1} |b_k|$
- $\forall j \leq i < h+1$: $sl_i = \sum_{k=j'}^{i} |b_k| \quad \Longrightarrow \quad j' \geq j$

We distinguish two cases:

(a) $\Delta l = 0$: $dl_{h+1} = \mathrm{base}(\alpha(PS), 0) + 2 + t = 1 + 2 + t = |b_{h+1}|$ This fulfills the conditions from the definition with $j = h+1$.

(b) $\Delta l > 0$: With the last corollary, we get that there is a $1 \leq m \leq h$ with

$$\mathrm{base}(\alpha(PS), \Delta l) = 1 + \sum_{k=m}^{h} |b_k|$$

Obviously, we have:

$$\begin{aligned} dl_{h+1} &= \mathrm{base}(\alpha(PS), \Delta l) + 2 + t \\ &= 1 + \sum_{k=m}^{h} |b_k| + 2 + t = \sum_{k=m}^{h} |b_k| + 3 + t = \sum_{k=m}^{h+1} |b_k| \end{aligned}$$

which fulfills condition (iii) of Definition 1 with $j = m$.
On the other hand, Corollary 2 states that:

$$\begin{aligned} dl_{h+1} &= \mathrm{base}(\alpha(PS), \Delta l) + 2 + t = \mathrm{base}(\alpha(PS), \Delta l - 1) + dl_h + 2 + t \\ &> dl_h \end{aligned}$$

and from this we can easily conclude condition (iv).

Finally, we have to prove, that $\mathrm{l}(\alpha'(PS)) = \mathrm{l}(\alpha(PS)) - \Delta l + 1$. We have

$$\begin{aligned} \mathrm{base}(\alpha'(PS), 1) &= 1 + \alpha'(PS) = 1 + dl_{h+1} \\ &= 1 + \mathrm{base}(\alpha(PS), \Delta l) + 2 + t \\ &= \mathrm{base}(\alpha(PS), \Delta l) + |b_{h+1}| \end{aligned}$$

This means, one step along the new link chain refers to Δl steps along the old chain plus a step along the newly created block, thus we can immediately finish the proof.

(ii) If $h > 1$ then the new value of PS is defined by deleting the topmost block, which obviously preserves the admissibility. q.e.d.

We now introduce a new notion, which enables us to relate the static structure of the program and the dynamic behavior of the procedure stack during the execution of the compiled stack program.

Definition 2 (declaration level) Let $P \in$ <u>`PLP-Prog`</u> and $Q = \Gamma_1, \ldots, \Gamma_q = \mathrm{trans}[\![P]\!]$ the translation of p. For every $3 \leq i \leq q$ exists $3 \leq a_i \leq i \leq b_i \leq q$, $B \in$ <u>`PLP-Block`</u>, $st \in$ <u>Tab</u> and $l \in \mathbb{N}_0$ with

$$\Gamma_{a_i}, \ldots, \Gamma_{b_i} = \mathrm{bt}[\![B]\!](st, l, a_i)$$

We define:

$$\mathrm{dl}(\Gamma_i) := \begin{cases} 0 & \text{if i=1,2} \\ l & \text{otherwise} \end{cases}$$

and we call $\mathrm{dl}(\Gamma_i)$ the **declaration level of the command Γ_i**. ◁

Corollary 4 Let $P \in$ <u>`PLP-Prog`</u>, $Q = \Gamma_1, \ldots, \Gamma_q = \mathrm{trans}[\![P]\!]$ and $1 \leq i \leq q$ such that $\Gamma_i =$ `STO` $(\Delta l, o)$ (`LOD`$(\Delta l, o)$, `CALL` $(ca, \Delta l, t)$). Then:

$$0 \leq \Delta l \leq \mathrm{dl}(\Gamma_i) \qquad \diamondsuit$$

Proof: trivial

We are able to show that the static structure of the program matches the dynamic structure of the stack if the procedure stack is admissible.

Theorem 1 Let $P \in$ <u>`PLP-Prog`</u>, $\mathfrak{A}_{\mathrm{SM}}(P)$ the stack machine for P, $\mathfrak{S} \in \underline{\mathrm{states}}_{\mathfrak{A}_{\mathrm{SM}}(P)}$ an initial state and $j \in \mathbb{N}$ with

$$\mathfrak{S} \vdash^{i}_{\mathfrak{A}_{\mathrm{SM}}(P)} \mathfrak{S}_i = <A, \alpha_i> \quad \forall 0 \leq i \leq j$$

with $\alpha_i(CODE)(\alpha_i(PC)) \neq$ `STO`$(\Delta l, o)$ $(\Delta l, o \in \mathbb{N}_0)$ for all $0 \leq i < j$ and $\alpha_i(PS)$ is admissible for all $0 \leq i \leq j$. Then we have:

$$\mathrm{dl}(\alpha_j(CODE)(\alpha_j(PC))) = \mathrm{l}(\alpha_j(PS)) \qquad \diamondsuit$$

Proof Induction on j:
$j = 0$: obvious

$j \to j+1$: We distinguish four cases:

(i) $\alpha_j(CODE)(\alpha_j(PC)) = \texttt{LIT}\ z$ ($\texttt{ADD}$, $\texttt{LOD}\ (\Delta l, o)$, etc.)
These commands do not change the value of PS, i.e.

$$\mathrm{l}(\alpha_{j+1}(PS)) = \mathrm{l}(\alpha_j(PS))$$

On the other hand, a block never ends with these commands, and we also have

$$\mathrm{dl}(\alpha_{j+1}(CODE)(\alpha_{j+1}(PC))) = \mathrm{dl}(\alpha_j(CODE)(\alpha_j(PC)))$$

Together with the induction hypothesis, we can conclude:

$$\mathrm{dl}(\alpha_{j+1}(CODE)(\alpha_{j+1}(PC))) = \mathrm{l}(\alpha_{j+1}(PS))$$

(ii) $\alpha_j(CODE)(\alpha_j(PC)) = \texttt{JMP}\ a$
If $\alpha_j(PC) > 2$ this is the same as case (i) since a is located within the same block. On the other hand, if $\alpha_j(PC) = 2$ then the declaration level is 0 by definition. From Lemma 2 we can conclude that the height of $\alpha_j(PS)$ is 1, because it has this height in the initial state $\mathfrak{S}$. The maximal chain length of an admissible stack with one block is also 0.

(iii) $\alpha_j(CODE)(\alpha_j(PC)) = \texttt{STO}\ (\Delta l, o)$
This is not possible.

(iv) $\alpha_j(CODE)(\alpha_j(PC)) = \texttt{CALL}\ (ca, \Delta l, t)$
With the last corollary and the induction hypothesis, we have

$$0 \leq \Delta l \leq \mathrm{dl}(\alpha_j(CODE)(\alpha_j(PC))) = \mathrm{l}(\alpha_j(PS))$$

Lemma 2 now guarantees that $\alpha_{j+1}(PS)$ is admissible and

$$\mathrm{l}(\alpha_{j+1}(PS)) = \mathrm{l}(\alpha_j(PS)) - \Delta l + 1$$

On the other hand, the command is created by the translation mapping, i.e. there are $ca, ac \in \mathbb{N}_0$, $\mathsf{P}_{\mathrm{dl}(\alpha_{j+1}(CODE)(\alpha_{j+1}(PC)))-1,o} \in \underline{\texttt{PIde}}$, $st, st' \in \underline{\mathrm{Tab}}$ and $B \in \underline{\texttt{PLP-Block}}$ with

(a) $st(\mathsf{P}_{\mathrm{dl}(\alpha_{j+1}(CODE)(\alpha_{j+1}(PC)))-1,o}) = (\texttt{proc}, ca, t)$
(b) $\alpha_j(CODE)(ca), \ldots, \alpha_j(CODE)(ac) = \mathrm{bt}[\![B]\!](st', \mathrm{dl}(\alpha_{j+1}(CODE)(\alpha_{j+1}(PC))), ca)$
(c) $\alpha_j(CODE)(\alpha_j(PC)) =$
$\mathrm{ct}[\![\mathsf{P}_{\mathrm{dl}(\alpha_{j+1}(CODE)(\alpha_{j+1}(PC)))-1,o}()]\!](st, \mathrm{dl}(\alpha_j(CODE)(\alpha_j(PC))), \alpha_j(PC))$

Thus we can conclude

$$\Delta l = \mathrm{dl}(\alpha_j(CODE)(\alpha_j(PC))) - \mathrm{dl}(\alpha_{j+1}(CODE)(\alpha_{j+1}(PC))) + 1$$

and this immediately leads to

$$\begin{aligned}
\mathrm{dl}(\alpha_{j+1}(CODE)(\alpha_{j+1}(PC))) &= \mathrm{dl}(\alpha_{j+1}(CODE)(\alpha_{j+1}(PC))) \\
&\quad + \mathrm{dl}(\alpha_j(CODE)(\alpha_j(PC))) + 1 \\
&\quad - \mathrm{dl}(\alpha_j(CODE)(\alpha_j(PC))) - 1 \\
&= \mathrm{dl}(\alpha_j(CODE)(\alpha_j(PC))) - \Delta l + 1 \\
&= \mathrm{l}(\alpha_j(PS)) - \Delta l + 1 \\
&= \mathrm{l}(\alpha_{j+1}(PS))
\end{aligned}$$

(v) $\alpha_j(CODE)(\alpha_j(PC)) = \texttt{RET}$
Because `STO` commands are not allowed, the stack can only be changed by `CALL` and `RET` commands. Thus there must be a maximal $0 \leq k < j$ with

$$\alpha_k(CODE)(\alpha_k(PC)) = \texttt{CALL}\ (ca, \Delta l, t)$$

for some $ca, \Delta l, t \in \mathbb{N}_0$. This `CALL` created the topmost block on the procedure stack. By the induction hypothesis, we get

$$\mathrm{dl}(\alpha_k(CODE)(\alpha_k(PC))) = \mathrm{l}(\alpha_k(PS))$$

Of course, the translation of a block never ends with a `CALL` command, i.e.

$$\mathrm{dl}(\alpha_{j+1}(CODE)(\alpha_{j+1}(PC))) = \mathrm{dl}(\alpha_k(CODE)(\alpha_k(PC)))$$

The `RET` command deletes the topmost block on the procedure stack, thus restoring the maximal chain length from $\alpha_k(PS)$):

$$\mathrm{l}(\alpha_{j+1}(PS)) = \mathrm{l}(\alpha_k(PS))$$

Putting this all together leads to the assertion. q.e.d.

Together with the last corollary this ensures that the *first* argument of a `STO` commands does not lead to an excess of the the maximal chain length. We still have to prove that the *second* argument of a `STO` command is all right. This is the content of the next theorem, which gives bounds for the dynamic links in an admissible stack.

Theorem 2 Let $P = \texttt{in/out}\texttt{X}_{0,1}, \ldots, \texttt{X}_{0,n}; B. \in \underline{\texttt{PLP-Prog}}$, $\mathfrak{A}_{\mathrm{SM}}(P)$ the stack machine for P, $\mathfrak{S} \in \underline{\mathrm{states}}_{\mathfrak{A}_{\mathrm{SM}}(P)}$ an initial state and $j \in \mathbb{N}$ with

$$\mathfrak{S} \vdash^{i}_{\mathfrak{A}_{\mathrm{SM}}(P)} \mathfrak{S}_i = <A, \alpha_i> \quad \forall 0 \leq i \leq j$$

with $\alpha_i(CODE)(\alpha_i(PC)) \neq \texttt{STO}(\Delta l, o)$ $(\Delta l, o \in \mathbb{N}_0)$ for all $0 \leq i < j$ and $\alpha_i(PS)$ is admissible for all $0 \leq i \leq j$. If we also have $\alpha_j(CODE)(\alpha_j(PC)) = \texttt{STO}(\Delta l, o)$ then:

$$1 \leq o \leq \alpha_j(PS).(\mathrm{base}(\alpha_j(PS), \Delta l) + 1) - 2$$

This means that the access of variables always hits the variable area of a block on the procedure stack. $\diamondsuit$

Proof With Theorem 1 and Corollary 4 we have $0 \leq \Delta l \leq \mathrm{l}(\alpha_j(PS))$. This means that

$$\alpha_j(PS).(\mathrm{base}(\alpha_j(PS), \Delta l) + 1)$$

is defined and is a dynamic link entry, i.e. $\alpha_j(PS).(\mathrm{base}(\alpha_j(PS), \Delta l) + 1) - 2 \geq 0$.
In the sequel, we will use the abbreviation $dl := \mathrm{dl}(\alpha_j(CODE)(\alpha_j(PC)))$. The stack code is created by the translation mapping, thus there exists a least surrounding block around an assignment which created the `STO` command. With other words there are $a, b, c, dl \in \mathbb{N}_0$, $\mathtt{X}_{l,o} \in \underline{\mathtt{VIde}}$, $st \in \underline{\mathrm{Tab}}$, $E \in \underline{\mathtt{PLP-Exp}}$ and $\Delta\Gamma \in \underline{\mathtt{PLP-Block}}$ with

(i) $3 \leq a \leq c < \alpha_j(PC) \leq b$

(ii) $\alpha_j(CODE)(a), \ldots, \alpha_j(CODE)(b) = \mathrm{bt}[\![B]\!](st, dl, a)$

(iii) $\alpha_j(CODE)(c), \ldots, \alpha_j(CODE)(\alpha_j(PC)) = \mathrm{ct}[\![\mathtt{X}_{l,o} := E]\!](st, dl, c)$

(iv) $st(\mathtt{X}_{l,o}) = (\mathtt{var})$

(v) $\Delta l = dl - l$

Because of (iv) there must be a sequence of symbol tables $st_0, \ldots, st_{dl} \in \underline{\mathrm{Tab}}$, a sequence of code addresses $a_1 \ldots, a_{dl} \in \mathbb{N}_0$, a sequence of declarations $\Delta_1 \ldots, \Delta_{dl} \in \underline{\mathtt{PLP-Decl}}$ and a sequence of commands $\Gamma_1 \ldots, \Gamma_{dl} \in \underline{\mathtt{PLP-Cmd}}$ with

- $st_{dl} = st$, $a_{dl} = a$ and $\Delta_{dl} = \Delta$.
- $st_{i+1} = \mathrm{upd}[\![\Delta_{i+1}]\!](st_i, i+1, a_{i+1})$ for all $0 \leq i < dl$

Obviously, the entry $st(\mathtt{X}_{l,o})$ is created in the update from st_{l-1} to st_l. We now distinguish two cases:

(i) $l = 0$, i.e. the variable is one of the I/O-variables. Obviously, we have

$$1 \leq o \leq n$$

and, according to the initial conditions I_{SM}, the first block of the procedure stack has the following structure:

$$b_1 = sl_1 : dl_1 : ra_1 : lv_{1,1} : \ldots, lv_{1,dl_1-2} = n+3 : n+2 : 0 : lv_{1,1} : \ldots, lv_{1,n}$$

Finally, we have

$$\begin{array}{rcl}\alpha_i(PS).(\mathrm{base}(\alpha_i(PS), \Delta l) + 1) - 2 & = & \alpha_i(PS).(\mathrm{base}(\alpha_i(PS), dl) + 1) - 2 \\ & = & \alpha_i(PS).(\mathrm{base}(\alpha_i(PS), \mathrm{l}(\alpha_i(PS))) + 1) - 2 \\ & = & dl_1 - 2 = n\end{array}$$

and this completes this case.

(ii) $l > 0$, i.e. $\mathtt{X}_{l,o}$ is a declared in Δ_l and we have:

$$0 \leq o \leq \text{size}[\![\Delta_l \Gamma_l]\!]$$

We still have to prove that

$$\text{size}[\![\Delta_l \Gamma_l]\!] = \alpha_i(PS).(\text{base}(\alpha_i(PS), \Delta l) + 1) - 2$$

Induction on Δl:
$\Delta l = 0$: We have $\text{base}(\alpha_i(PS), \Delta l) = \text{base}(\alpha_i(PS), 0) = 1$, i.e.

$$\alpha_i(PS).(\text{base}(\alpha_i(PS), \Delta l) + 1) = \alpha_i(PS).2$$

Assume, that the height of $\alpha_i(PS)$ is less than 2. Then we immediately have

$$\text{l}(\alpha_i(PS)) = dl = 0$$

and thus $l = 0$. This is a contradiction to the assumption of this case, we now can conclude, that the height of $\alpha_i(PS)$ is at least 2. This means that the last `CALL` command created the topmost block on the procedure stack and this means that

$$\text{size}[\![\Delta_l \Gamma_l]\!] + 2 = \alpha_i(PS).2$$

which concludes this part of the proof.

$\Delta l \to \Delta l + 1$: Let

$$p' = \alpha_i(PS).(\text{base}(\alpha_i(PS), 1)) : \ldots : \alpha_i(PS).|\alpha_i(PS)|$$

With Corollary 2 and the induction hypothesis we have:

$$\begin{aligned}
\alpha_i(PS).(\text{base}(\alpha_i(PS), \Delta l + 1) + 1) - 2 &= \alpha_i(PS).(\text{base}(p', \Delta l) + \alpha_i(PS).1 + 1) - 2 \\
&= \alpha_i(PS).(\text{base}(\alpha_i(PS), 1) + \text{base}(p', \Delta l) + 1) - 2 \\
&= p'.(\text{base}(p', \Delta l) + 1) - 2 \\
&= \text{size}[\![\Delta_l \Gamma_l]\!]
\end{aligned}$$

q.e.d.

Now that we know that the `STO` commands is save, we can give the main result of this section:

Theorem 3 Let $P \in \underline{\mathtt{PLP\text{-}Prog}}$, $\mathfrak{A}_{\text{SM}}(P)$ the stack machine for P, $<A, \alpha> \in \underline{\text{states}}_{\mathfrak{A}_{\text{SM}}(P)}$ a reachable state. Then:

$$\alpha(PS) \text{ is admissible}$$

I.e. during the execution of a compiled program, the procedure stack stays admissible. $\diamondsuit$

Proof Induction on j, the number of steps to reach state $<A, \alpha>$:
$j = 0$: From the initial conditions I_{SM} we can deduce that there are $z, z_1, \ldots, z_{\alpha_0(arity)} \in \mathbb{Z}$ with:

$$\alpha_0(PS) = \alpha_0(arity) + 3 : \alpha_0(arity) + 2 : z : z_1 : \ldots : z_{\alpha_0(arity)}$$

Obviously, this is an admissible procedure stack.

$j \to j+1$: With the induction hypothesis we get that $\alpha_j(PS)$ is admissible. Now, we distinguish which rule is applicated during the step $\mathfrak{S}_j \vdash_{\mathfrak{A}_{\mathrm{SM}(P)}} \mathfrak{S}_{j+1}$. Of course, we only have to take these rules into account, which modify PS.

(i) $\alpha_j(CODE)(\alpha_j(PC)) = \mathtt{CALL}\ (ca, \Delta l, t)$. Corollary 4 ensures that $\Delta l \leq \mathrm{l}(\alpha_j(PS))$ thus we can apply Lemma 2 and this leads to the claim.

(ii) $\alpha_j(CODE)(\alpha_j(PC)) = \mathtt{RET}$. Again, we want to apply Lemma 2. Assume that the height of $\alpha_j(PS)$ is less than 2. This immediately leads to $\mathrm{l}(\alpha_j(PS)) = 0$, i.e.

$$\mathrm{dl}(\alpha_j(CODE)(\alpha_j(PC))) = 0$$

which is impossible.

(iii) $\alpha_j(CODE)(\alpha_j(PC)) = \mathtt{STO}(\Delta l, o)$. Corollary 4 and the last theorem guarantees that $\alpha_{j+1}(PS)$ is defined and admissible, because static link and dynamic link entries stay unchanged.

q.e.d.

In this section we proved that the procedure stack stays admissible during execution of compiled PLP programs. We will exploit this result in the next section to classify the type of information stored in a procedure stack and to define a modified machine, which separates the procedure stack into several parts.

6 The modified machine $\mathfrak{A}_{\mathrm{SM_m}}$

An admissible procedure stack can be interpreted as an amalgamation of three kinds of information:

(i) a *current state* χ which holds the bindings of those variables accessible by the STO and LOD commands,

(ii) a *stack of variables bindings* s which are protected from access through the static links, and, of course,

(iii) a *stack of return addresses* r.

Until now the variables were accessed *relative* to the top of the procedure stack. From now on we will use *absolute* access w.r.t. the bottom of the stack.

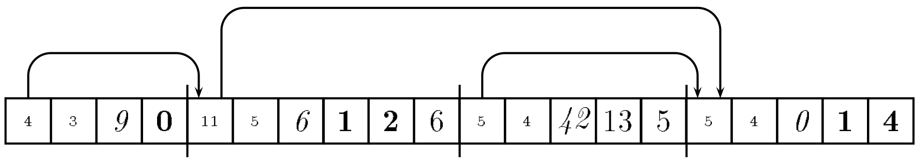
We then choose the parts to be of the following types:

(i) $\chi \in \underline{S}$, where $\chi(\mathtt{X}_{l,o})$ represents the value which can be accessed via $\mathtt{STO}(\mathrm{l}(p) - l, o)$ and $\mathtt{LOD}(\mathrm{l}(p) - l, o)$.

(ii) $s \in (\mathbb{N} \times \mathbb{N} \times \underline{S})^\star$, a list where each entry has the form (l, t, χ'). This information is used to restore the value of χ during execution of `RET` commands. The first component denotes that all variables $\mathtt{X}_{l,i}$ in χ will be unbound and then overwritten by the bindings in the third component χ'. We could drop l if we can ensure that χ' is never empty. In this case l can be extracted from χ' because all bindings in χ' affect only variables of this level. The second component holds the number of local variables in the current block. It is for technical reasons only; the machine does not use it. We will need it in order to prove the equivalence of the machines.

(iii) $r \in \mathbb{N}^\star$ representing a stack of return addresses.

Recall the example from Section 3:



$\chi =$ $[\mathtt{X}_{2,1}/\mathbf{0}]$ $[\mathtt{X}_{1,1}/\mathbf{1},\ \mathtt{X}_{1,2}/\mathbf{2},\ \mathtt{X}_{1,3}/\mathbf{6}]$ $[\mathtt{X}_{0,1}/\mathbf{0},\ \mathtt{X}_{0,2}/\mathbf{1},\ \mathtt{X}_{0,3}/\mathbf{4}]$

$s =$ $(2, 1, [])(1, 3, [\mathtt{X}_{1,1}/13,\ \mathtt{X}_{1,2}/5])$ $(1, 2, [])$ $(0, 2, [])$

$r =$ $\mathit{9}$ $\mathit{6}$ $\mathit{42}$ $\mathit{0}$

If the topmost block of p is deleted by a `RET` command, we get p' with the associated parts:

- $\chi' = [\mathtt{X}_{1,1}/1,\ \mathtt{X}_{1,2}/2,\ \mathtt{X}_{1,3}/6,\ \mathtt{X}_{0,1}/0,\ \mathtt{X}_{0,2}/1, \mathtt{X}_{0,3}/4]$, i.e. the bindings of variables of level 2 were deleted by application of the topmost entry of s,
- $s' = (1, 3, [\mathtt{X}_{1,1}/13,\ \mathtt{X}_{1,2}/5]) : (1, 2, []) : (0, 2, [])$ and
- $r' = 6 : 42 : 0$

We now introduce the modified stack machine:

- The new commands `GETX` and `PUTX` replace `STO` resp. `LOD`.
  <u>`SCm-Inst`</u> Γ::= `LIT` z | `ADD` | `JMP` a | `JMNC` a |
  `GETX` (l, o) | `PUTX` (l, o) | `CALL` (ca, l, t) | `RET`
  <u>`SCm-Prog`</u> Q::=$\Gamma_1, \ldots, \Gamma_n$
- The signature is changed appropriately to $\Sigma_{\mathrm{SM_m}} = (S_{\mathrm{SM_m}}, \Omega_{\mathrm{SM_m}})$:
  - Sorts $S_{\mathrm{SM_m}} := \{\underline{int}, \underline{intstack}, \underline{scminst}, \underline{varbind}, \underline{savestack}\}$

– Operation symbols $\Omega_{\mathrm{SM_m}} := \Omega_{\mathrm{SM_m},s} \dot\cup \Omega_{\mathrm{SM_m},d}$

$$
\begin{array}{rcl}
\Omega_{\mathrm{SM_m},s} & := & \{\ \mathit{CODE}\ (\underline{int}, \underline{scminst}),\ \mathit{arity}\ (\underline{int}),\\
& & \mathit{0}\ (\underline{int}),\ \mathit{1}\ (\underline{int}),\ +\ (\underline{int}\cdot\underline{int}, \underline{int}),\ \epsilon\ (\underline{intstack}),\ \epsilon\ (\underline{savestack}),\\
& & \mathtt{LIT}\ (\underline{int}, \underline{scminst}),\ \mathtt{ADD}\ (\underline{scminst}),\\
& & \mathtt{JMP}\ (\underline{int}, \underline{scminst}),\ \mathtt{JMNC}\ (\underline{int}, \underline{scminst}),\\
& & \mathtt{PUTX}\ (\underline{int}\cdot\underline{int}, \underline{scminst}),\ \mathtt{GETX}\ (\underline{int}\cdot\underline{int}, \underline{scminst}),\\
& & \mathtt{CALL}\ (\underline{int}\cdot\underline{int}\cdot\underline{int}, \underline{scminst}),\ \mathtt{RET}\ (\underline{scminst})\\
& & \mathit{push}\ (\underline{intstack}\cdot\underline{int}, \underline{intstack}),\ \mathit{pop}\ (\underline{intstack}, \underline{intstack}),\\
& & \mathit{top}\ (\underline{intstack}, \underline{int}),\\
& & \mathit{read}\ (\underline{varbind}\cdot\underline{int}\cdot\underline{int}, \underline{int}),\\
& & \mathit{write}\ (\underline{varbind}\cdot\underline{int}\cdot\underline{int}\cdot\underline{int}, \underline{varbind}),\\
& & \mathit{clear}\ (\underline{varbind}\cdot\underline{int}, \underline{varbind}).\\
& & \mathit{save}\ (\underline{savestack}\cdot\underline{int}\cdot\underline{int}\cdot\underline{varbind}, \underline{savestack}),\\
& & \mathit{restore}\ (\underline{savestack}\cdot\underline{int}\cdot\underline{varbind}, \underline{varbind}),\\
& & \mathit{defnd}\ (\underline{varbind}\cdot\underline{int}, \underline{int}),\\
& & \mathit{ssinit}\ (\underline{savestack}),\\
& & \mathit{pop}\ (\underline{savestack}, \underline{savestack})\}\\
\Omega_{\mathrm{SM_m},d} & := & \{\ \mathit{PC}\ (\underline{int}),\ \mathit{DS}\ (\underline{intstack}),\\
& & \mathit{CHI}\ (\underline{varbind}),\ \mathit{SS}\ (\underline{savestack}),\ \mathit{RS}\ (\underline{intstack})\}
\end{array}
$$

The new symbols *CHI*, *SS* and *RS* replace the procedure stack in the original machine. Bindings of variables in *CHI* are accessed by *read* and *write*, and the actions on procedure call and procedure exit are performed by *clear*, *save* and *restore*. The symbols *defnd* and *ssinit* are used for the initial condition only.

- The carrier of the EA is

$$A_{\mathrm{SM_m}} = (A^{\underline{int}}, A^{\underline{intstack}}, A^{\underline{scminst}}, A^{\underline{varbind}}, A^{\underline{savestack}})$$

with

– $A^{\underline{int}} := \mathbb{Z}, A^{\underline{intstack}} := \mathbb{Z}^\star,\ A^{\underline{scminst}} := \underline{\mathtt{SC_m\text{-}Inst}}$,

– $A^{\underline{varbind}} := \underline{\mathrm{S}}$ and

– $A^{\underline{savestack}} := (\mathbb{N}_0 \times \mathbb{N}_0 \times A^{\underline{varbind}})^\star$

- The interpretation of the new static symbols $\varphi_{\mathrm{SM_m}}$:

$$
\begin{array}{rcl}
\varphi_{\mathrm{SM_m}}(\epsilon) & := & \varepsilon\\
\varphi_{\mathrm{SM_m}}(\mathtt{GETX})(l,o) & := & \mathtt{GETX}\ (l,o)\\
\varphi_{\mathrm{SM_m}}(\mathtt{PUTX})(l,o) & := & \mathtt{PUTX}\ (l,o)
\end{array}
$$

$$
\begin{array}{rcl}
\varphi_{\mathrm{SM_m}}(\mathtt{CALL})(ca, l, t) & := & \mathtt{CALL}\ (ca, l, t) \\
\varphi_{\mathrm{SM_m}}(read)(\chi, l, o) & = & \chi(\mathtt{X}_{l,o}) \\
\varphi_{\mathrm{SM_m}}(write)(\chi, l, o, z) & = & \chi[\mathtt{X}_{l,o}/z] \\
\varphi_{\mathrm{SM_m}}(clear)(\chi, l, n) & = & \chi[\mathtt{X}_{l,1}/0, \ldots, \mathtt{X}_{l,n}/0] \\
\varphi_{\mathrm{SM_m}}(save)(s, l, t, \chi) & = & (l, t, \chi_l) : s \\
\varphi_{\mathrm{SM_m}}(restore)((l, t, \chi) : s, \chi') & = & \mathrm{reset}(\chi, l, \chi') \\
\varphi_{\mathrm{SM_m}}(defnd)(\chi, l) & = & \max\{o \in \mathbb{N} \mid\mid \chi(l, o) \text{ defined}\} \\
\varphi_{\mathrm{SM_m}}(ssinit) & = & \varepsilon
\end{array}
$$

We use the same auxiliary function 'reset', which we already know from the denotational semantics of `PLP`. The definition of *CODE* and *arity* is missing again.

- The rules $\mathcal{R}_{\mathrm{SM_m}}$ are the same as the rules $\mathcal{R}_{\mathrm{SM}}$ except those for `STO`, `LOD`, `CALL` and `RET`, plus new rules for `GETX`, `PUTX`, `CALL` and `RET`:

  if $CODE(PC) = \mathtt{GETX}(l, o)$ **then**
  $\quad PC := PC + 1$
  $\quad DS := push(DS, read(CHI, l, o))$
  fi,
  if $CODE(PC) = \mathtt{PUTX}(l, o)$ **then**
  $\quad PC := PC + 1$
  $\quad CHI := write(CHI, l, o, top(DS)))$
  $\quad DS := pop(DS)$
  fi,
  if $CODE(PC) = \mathtt{CALL}(ca, l, t)$ **then**
  $\quad PC := ca$
  $\quad RS := push(RS, PC + 1)$
  $\quad SS := save(SS, l, t, CHI)$
  $\quad CHI := clear(CHI, l, t)$
  fi,
  if $CODE(PC) = \mathtt{RET}$ **then**
  $\quad PC := top(RS)$
  $\quad RS := pop(RS)$
  $\quad SS := pop(SS)$
  $\quad CHI := restore(SS, CHI)$
  fi

- Initial and final conditions:

$$
\begin{array}{rcl}
I_{\mathrm{SM_m}} & := & \{PC = 1,\ SS = ssinit,\ RS = push(\epsilon, 0),\ defnd(CHI, 0) = arity\} \\
F_{\mathrm{SM_m}} & := & \{PC = 0\}
\end{array}
$$

- The modified translation mapping $\mathrm{trans_m}$: <u>`PLP-Prog`</u> $\dashrightarrow$ <u>$\mathtt{SC_m}$`-Prog`</u> (and, of course, '$\mathrm{bt_m}$', '$\mathrm{dt_m}$' etc.) are the same as the original ones, except that

(i) $\texttt{LOD}(l-dl,o)$, $\texttt{STO}(l-dl,o)$ and $\texttt{CALL}(ca,l-dl,t)$ are replaced by $\texttt{GETX}(dl,o)$, $\texttt{PUTX}(dl,o)$ and $\texttt{CALL}(ca,dl,t)$

(ii) $\texttt{CALL}(3,0,\mathrm{size}[\![B]\!])$ is replaced by $\texttt{CALL}(3,1,\mathrm{size}[\![B]\!])$

With this mappings we can fill in $\varphi_{\mathrm{SM_m}}(CODE)$ and $\varphi_{\mathrm{SM_m}}(arity)$ exactly the same way, thus we can define $\mathfrak{A}_{\mathrm{SM_m},P}$ for a given PLP-program P.

The next aim is to prove that $\mathfrak{A}_{\mathrm{SM}}(P)$ and $\mathfrak{A}_{\mathrm{SM_m}}(P)$ are equivalent.

7 Equivalence of $\mathfrak{A}_{\mathrm{SM}}(P)$ and $\mathfrak{A}_{\mathrm{SM_m}}(P)$

Let $P \in \underline{\texttt{PLP-Prog}}$ be an arbitrary program. Recall the definition of equivalence of EA's from Section 2:

$\mathfrak{A}_{\mathrm{SM}}(P)$ is equivalent to $\mathfrak{A}_{\mathrm{SM_m}}(P)$ iff $\mathfrak{A}_{\mathrm{SM}}(P)$ is sound and complete w.r.t. $\mathfrak{A}_{\mathrm{SM_m}}(P)$.

We will prove both directions of this equivalence separately. Each direction requires the definition of a function h, which converts the reachable states of the machines.

In order to prove the soundness, we will have to define an auxiliary function first. We define the set PS_a to be the set of all admissible stacks plus the empty stack:

$$PS_a := \{p \in \mathbb{Z}^\star \mid\mid p = \varepsilon \text{ or } p \text{ admissible}\}$$

Now let $\mathfrak{S},\mathfrak{T} \in \underline{\mathrm{states}}_{\mathfrak{A}_{\mathrm{SM}}(P)}$ be states with $\mathfrak{S} = <A, \alpha_{\mathfrak{S}}>$ initial and $\mathfrak{T} = <A, \alpha_{\mathfrak{T}}>$ final. The main result of Section 5 was Theorem 3, which ensures that $\alpha_{\mathfrak{S}}(PS)$ and $\alpha_{\mathfrak{T}}(PS)$ are admissible. This means that it is sufficient to define a function on the set PS_a, which creates an appropriate representation of the procedure stack in the modified machine, i.e. as element of $(\mathbb{Z}^\star \times \underline{\mathrm{S}} \times (\mathbb{N}_0 \times \mathbb{N}_0 \times \underline{\mathrm{S}})^\star)$, and to extend it to the desired function h later. We define the function

$$\mathrm{decode} : PS_a \to (\mathbb{Z}^\star \times \underline{\mathrm{S}} \times (\mathbb{N}_0 \times \mathbb{N}_0 \times \underline{\mathrm{S}})^\star)$$

by induction on the number of blocks of the procedure stack:

$$\begin{array}{rcl} \mathrm{decode}(\varepsilon) & := & (\varepsilon, [], \varepsilon) \\ \mathrm{decode}(\underbrace{sl : dl : ra : lv_1 : \ldots : lv_{dl-2} : p}_{p'}) & := & (ra : r, \chi[\texttt{X}_{l,1}/lv_1 \ldots, \texttt{X}_{l,dl-2}/lv_{dl-2}], (l, dl-2, \chi_l) : s) \\ & & \text{if } (r,\chi,s) = \mathrm{decode}(p) \text{ and } l = \mathrm{l}(p') \end{array}$$

The way the function works is quite the same, as the CALL command in the modified machine, except that it does not bind the new variables to zero, but to the value found in the procedure stack. Computing $\mathrm{decode}(p)$ means simulating a sequence of CALL commands and assignments in the modified machine.

It is quite obvious that this function is injective; two different elements p_1 and p_2 of PS_a have either a different static/dynamic-link structure or the same structure and different return addresses or variable bindings. In both cases $\mathrm{decode}(p_1)$ will be different from $\mathrm{decode}(p_2)$, because some-when in the computation of 'decode' there will be a different dl or $\mathrm{l}(p)$ in the first case and a different ra or lv_i in the second case.

Lemma 3 Let $p \in \mathbb{Z}^\star$ be an admissible procedure stack, $(r, \chi, s) = \text{decode}(p)$ and $\Delta l, o \in \mathbb{N}_0$ with $\Delta l \leq \text{l}(p)$ and $1 \leq o \leq p.(\text{base}(p, \Delta l) + 1) - 2$. Then we have:

$$\chi(\mathtt{X}_{\text{l}(p)-\Delta l, o}) = p.(\text{base}(p, \Delta l) + o + 2) \qquad \diamondsuit$$

Proof The restriction of o ensures that the right hand side hits a variable entry, thus the left hand side is defined. Because p is admissible, it has the form:

$$p = sl : dl : ra : lv_1 : \ldots : lv_{dl-2} : p'$$

Let $\text{decode}(p') = (r', \chi', s')$ and $l = \text{l}(p)$.
Induction on Δl:
$\Delta l = 0$: From the definition of 'base', we have:

$$\text{base}(p, \Delta l) = \text{base}(p, 0) = 1$$

and thus

$$p.(\text{base}(p, \Delta l) + o + 2) = p.(o + 3) = lv_o$$

On the other hand, χ is constructed by

$$\chi = \chi'[X_{l,1}/lv_1 \ldots, X_{l,dl-2}/lv_{dl-2}]$$

and obviously

$$\chi(\mathtt{X}_{\text{l}(p)-\Delta l, o}) = \chi(\mathtt{X}_{\text{l}(p), o}) = \chi(\mathtt{X}_{l,o}) = lv_o$$

$\Delta l \to \Delta l + 1$: Corollary 2 now yields:

$$\text{base}(p, \Delta l + 1) = \text{base}(p', \Delta l) + p.1$$

This means that the expression on the right hand side evaluates to

$$\begin{aligned} p.(\text{base}(p, \Delta l + 1) + o + 2) &= p.(\text{base}(p', \Delta l) + p.1 + o + 2) \\ &= p'(\text{base}(p', \Delta l) + o + 2) \end{aligned}$$

On the other hand, we have $\text{l}(p') = \text{l}(p) - 1$ and thus:

$$\chi(\mathtt{X}_{\text{l}(p)-(\Delta l+1), o}) = \chi(\mathtt{X}_{\text{l}(p)-1-\Delta l, o}) = \chi'(\mathtt{X}_{\text{l}(p')-\Delta l, o})$$

The induction hypothesis now completes the proof. q.e.d.

Theorem 4 (Soundness) Let $P \in \underline{\texttt{PLP-Prog}}$ be a program. Then:

$$\mathfrak{A}_{\mathrm{SM}}(P) \text{ is sound w.r.t. } \mathfrak{A}_{\mathrm{SM_m}}(P) \qquad \diamondsuit$$

Proof We have to prove that there is an injective function $h : \underline{\text{states}}_{\mathfrak{A}_{\mathrm{SM}}(P)} \dashrightarrow \underline{\text{states}}_{\mathfrak{A}_{\mathrm{SM_m}}(P)}$ such that $\forall \mathfrak{S}, \mathfrak{T} \in \underline{\text{states}}_{\mathfrak{A}_{\mathrm{SM}}(P)}$, with $\mathfrak{S}$ initial and $\mathfrak{T}$ final, holds:

$$\mathfrak{S} \vdash^{\star}_{\mathfrak{A}_{\mathrm{SM}}(P)} \mathfrak{T} \quad \Longrightarrow \quad h(\mathfrak{S}) \vdash^{\star}_{\mathfrak{A}_{\mathrm{SM_m}}(P)} h(\mathfrak{T}),\ h(\mathfrak{S}) \text{ initial and } h(\mathfrak{T}) \text{ final}$$

Using the auxiliary function 'decode', we can define the function h by

- applying *decode* to the value of PS, thus getting values for RS, CHI and SS,
- transferring the values of DS and PC unchanged, and
- mapping the (fixed) interpretation of the static symbols of the original machine to the (fixed) interpretation of the modified machine.

In the sequel, let $\mathfrak{S}, \mathfrak{T} \in \underline{\text{states}}_{\mathfrak{A}_{\mathrm{SM}}(P)}$ be states with $\mathfrak{S} = <A, \alpha_{\mathfrak{S}}>$ initial and $\mathfrak{T} = <A, \alpha_{\mathfrak{T}}>$ final.

First of all, we are going to prove, that h respects initial and final states. We have to prove that

(i) $h(\mathfrak{S})$ is an initial state, and

(ii) $h(\mathfrak{T})$ is a final state.

Let $P = \texttt{in/out}\ \texttt{X}_{0,1}, \ldots, \texttt{X}_{0,n}; B.$, i.e. $\alpha_{\mathfrak{S}}(arity) = n$.
(i): The initial condition I_{SM} ensures that

$$\alpha_{\mathfrak{S}}(PS) = n+3 : n+2 : 0 : z_1 : \ldots : z_n$$

If we apply 'decode' to $\alpha_{\mathfrak{S}}(PS)$ we get:

$$\text{decode}(\alpha_{\mathfrak{S}}(PS)) = (0, [\texttt{X}_{0,1}/z_1, \ldots, \texttt{X}_{0,n}/z_n], (0, []))$$

and thus

$$h(\alpha_{\mathfrak{S}})(RS) = 0, \quad h(\alpha_{\mathfrak{S}})(CHI) = [\texttt{X}_{0,1}/z_1, \ldots, \texttt{X}_{0,n}/z_n], \quad h(\alpha_{\mathfrak{S}})(RS) = (0, [])$$

which is exactly the initial condition $I_{\mathrm{SM_m}}$.
(ii) is trivial.

Now we have to prove that $\mathfrak{S} \vdash^{\star}_{\mathfrak{A}_{\mathrm{SM}}(P)} \mathfrak{T}$ implies that $h(\mathfrak{S}) \vdash^{\star}_{\mathfrak{A}_{\mathrm{SM_m}}(P)} h(\mathfrak{T})$. We will prove the stronger condition: if $\mathfrak{S}_i = <A, \alpha_i> \in \underline{\text{states}}_{\mathfrak{A}_{\mathrm{SM}}}$ $(i = 1, 2)$ then:

$$\mathfrak{S}_1 \vdash_{\mathfrak{A}_{\mathrm{SM}}(P)} \mathfrak{S}_2 \quad \Longrightarrow \quad h(\mathfrak{S}_1) \vdash_{\mathfrak{A}_{\mathrm{SM_m}}(P)} h(\mathfrak{S}_2)$$

Now let the condition $\mathfrak{S}_1 \vdash_{\mathfrak{A}_{\mathrm{SM}}(P)} \mathfrak{S}_2$ be true and let $<B, \beta_1> = h(\mathfrak{S}_1)$. Obviously, the modified machine can do a uniquely defined step $h(\mathfrak{S}_1) \vdash_{\mathfrak{A}_{\mathrm{SM_m}}(P)} <B, \beta_2>$. We have to prove that

$$\text{decode}(\alpha_2(PS)) = (\beta_2(RS), \beta_2(CHI), \beta_2(SS))$$

which will imply, that $<B, \beta_2> = h(\mathfrak{S}_1)$. For this aim, we have to distinguish which rule is used; this depends on $\Gamma := \alpha_1(CS)(\alpha_1(PC))$:

(i) $\Gamma = \texttt{LIT}(z)$ ($\texttt{ADD}$, $\texttt{JMP}(a), \ldots$)
this is trivial, because $\beta_1(CODE)(\beta_1(PC)) = \Gamma$ and the rules are the same in both machines.

(ii) $\Gamma = \texttt{LOD}(\Delta l, o)$
This means, that

$$\beta_1(CODE)(\beta_1(PC)) = \texttt{GETX}(\mathrm{dl}(\Gamma) - \Delta l, o)$$

The selected number is pushed onto the data stack in both cases; thus with $p := \alpha_1(PS)$ and $\chi := \beta_1(CHI)$ we have to prove, that:

$$\chi(\texttt{X}_{\mathrm{dl}(\Gamma)-\Delta l, o}) = p.(\mathrm{base}(p, \Delta l) + o + 2)$$

Remember from Section 5, that $l = \mathrm{dl}(\Gamma)$. Now we can finish this case with Lemma 3.

(iii) $\Gamma = \texttt{STO}(\Delta l, o)$
Analogous to the last case.

(iv) $\Gamma = \texttt{CALL}(ca, \Delta l, t)$
With $l := \mathrm{dl}(\Gamma) - \Delta l$ the matching command in $\mathfrak{A}_{\mathrm{SM_m}}(P)$ is

$$\beta_1(CODE)(\beta_1(PC)) = \texttt{CALL}(ca, l, t)$$

and with $p := \alpha_1(PS)$, $r = \beta_1(RS)$, $\chi := \beta_1(CHI)$, and $s := \beta_1(SS)$ we have:

(a) $p' := \alpha_2(PS) = (\mathrm{base}(p, \Delta l) + 2 + t) : (t + 2) : (\alpha_1(PC) + 1) : \underbrace{0 : \ldots : 0}_{t} : p$
(b) $r' := \beta_2(RS) = (\beta_1(PC) + 1) : r$
(c) $\chi' := \beta_2(CHI) = \chi[\texttt{X}_{l,1}/0, \ldots, \texttt{X}_{l,t}/0]$
(d) $s' := \beta_2(SS) = (l, t, \chi_l) : s$

and this concludes this case.

(v) $\Gamma = \texttt{RET}$
With

$$p := \alpha_1(PS) = sl : dl : ra : lv_1 : \ldots : lv_{dl-2} : p'$$

and $l = \mathrm{l}(p)$ we have

(a) $\beta_1(RS) = ra : r'$,
(b) $\beta_1(SS) = (l, dl - 2, \chi'_l) : s'$
(c) $\beta_1(CHI) = \chi'[\texttt{X}_{l,1}/lv_1, \ldots, \texttt{X}_{l,dl-2}/lv_{l,dl-2}]$, and

and, of course,

$$\mathrm{decode}(p') = (r', \chi', s')$$

Thus the values in the next states are:

(a) $\alpha_2(PS) = p'$,
(b) $\beta_2(RS) = r'$
(c) $\beta_2(SS) = s'$
(d) $\beta_2(CHI) = \mathrm{reset}(\chi'_l, l, \beta_1(CHI)) = \mathrm{reset}(\chi'_l, l, \chi'[\texttt{X}_{l,1}/lv_1, \ldots, \texttt{X}_{l,dl-2}/lv_{l,dl-2}]) = \chi'$

and this concludes this case and also the proof. q.e.d.

In order to do the other direction we have to define an other auxiliary function, which is essentially the inverse function to decode. We define the function

$$\text{encode} : (\mathbb{Z}^\star \times \underline{S} \times (\mathbb{N}_0 \times \mathbb{N}_0 \times \underline{S})^\star) \dashrightarrow PS_a$$

by induction on the length of the stacks:

$$\begin{array}{rcl} \text{encode}(\varepsilon, [], \varepsilon) & := & (\varepsilon) \\ \text{encode}(ra : r, \chi, (l, t, \chi') : s) & := & sl : dl : ra : lv_1 : \ldots : lv_{dl-2} : p \\ & & \text{if } p = \text{encode}(r, \text{reset}(\chi', l, \chi), s), \\ & & \quad sl = \begin{cases} 1 + dl & \text{if } p = \varepsilon \\ \text{base}(p, \text{l}(p) - l + 1) + dl & \text{otherwise} \end{cases} \\ & & \quad dl = t - 2, \\ & & \quad lv_i = \chi(\mathtt{X}_{l,i}) \qquad \forall 1 \leq i \leq dl - 2, \end{array}$$

It simulates a $\mathtt{CALL}(., \text{l}(p) - l + 1, t)$ sequence of the original machine. Thus it is obvious (recall Lemma 2), that the result of 'encode' is always admissible (or ε) if it is defined. This function is not defined, if the length of its first argument is not the same as the length of its third argument. At this point it becomes clean, why there is the additional second component t in the topmost entry of SS. Remember that the modified machine did not use it. The only reason for its introduction was the definition of this function, i.e. to make it possible to reproduce the procedure stack. This information can not be derived from χ or χ', because there may be entries in these environments, which do not correspond to a local variable in the current block, but to one in another block of the same level. Note that when the modified machine performs a `CALL`, it only overwrites the binding of the new local variables, but does not delete the bindings of other variables of the same level.

Lemma 4 Let $p \in PS_a$, $r \in \mathbb{Z}^\star$, $\chi \in \underline{S}$ and $s \in (\mathbb{N}_0 \times \mathbb{N}_0 \times \underline{S})^\star$. Then we have:

$$\text{encode}(r, \chi, s) = p \quad \Longleftrightarrow \quad (r, \chi, s) = \text{decode}(p) \qquad \diamondsuit$$

Proof Induction on the number of blocks of p:
$p = \varepsilon$: trivial
$p = sl : dl : ra : lv_1 : \ldots : lv_{dl-2} : p'$: Let $(r', \chi', s') = \text{decode}(p')$ and $l = l(p)$. Thus:

$$\begin{array}{rcl} r & = & ra : r' \\ \chi & = & \chi'[\mathtt{X}_{l,1}/lv_1 \ldots, \mathtt{X}_{l,dl-2}/lv_{dl-2}] \\ s & = & (l, dl - 2, \chi'_l) : s' \end{array}$$

By the induction hypothesis we have $\text{encode}(r', \chi', s') = p'$, thus:

$$\begin{array}{rcl} \text{encode}(r, \chi, s) & = & \text{encode}(ra : r', \chi'[\mathtt{X}_{l,1}/lv_1 \ldots, \mathtt{X}_{l,dl-2}/lv_{dl-2}], (l, dl - 2, \chi'_l) : s')) \\ & = & sl' : dl : ra : lv_1 : \ldots : lv_{dl-2} : p' \\ & & \text{with } sl = \begin{cases} 1 + dl & \text{if } p' = \varepsilon \\ \text{base}(p', \text{l}(p') - l + 1) + dl & \text{otherwise} \end{cases} \end{array}$$

This means that we only have to prove that $sl = sl'$. We distinguish two cases:

(i) $p' = \varepsilon$. This means that p has one block, i.e. $sl = dl + 1 = sl'$.

(ii) $p' \neq \varepsilon$. In this case 'encode' simulates a $\texttt{CALL}(., \mathrm{l}(p') - l + 1, t)$ of the original machine. Lemma 2 now ensures, that

$$\begin{aligned} \mathrm{l}(sl' : dl : ra : lv_1 : \ldots : lv_{dl-2} : p') &= \mathrm{l}(p') - (\mathrm{l}(p') - l + 1) + 1 \\ &= \mathrm{l}(p') - \mathrm{l}(p') + \mathrm{l}(p) \\ &= \mathrm{l}(p) \end{aligned}$$

and this immediately leads to $sl = sl'$. q.e.d.

Theorem 5 (Completeness) Let $P \in$ <u>`PLP-Prog`</u> be a program. Then:

$$\mathfrak{A}_{\mathrm{SM}}(P) \text{ is complete w.r.t. } \mathfrak{A}_{\mathrm{SM_m}}(P) \qquad \diamondsuit$$

Proof We have to show that $\mathfrak{A}_{\mathrm{SM_m}}(P)$ is sound w.r.t. $\mathfrak{A}_{\mathrm{SM}}(P)$, thus we will proceed in the same way as in the last theorem, just replacing 'decode' by 'encode'. By doing so, we obtain a function $g : \underline{\mathrm{states}}_{\mathfrak{A}_{\mathrm{SM_m}}(P)} \dashrightarrow \underline{\mathrm{states}}_{\mathfrak{A}_{\mathrm{SM}}(P)}$ which respects initial and final states. All we have to prove is that if $<B, \beta_i> \in \underline{\mathrm{states}}_{\mathfrak{A}_{\mathrm{SM_m}}(P)}$ (for $i = 1, 2$) then:

$$<B, \beta_1> \vdash_{\mathfrak{A}_{\mathrm{SM_m}}(P)} <B, \beta_2> \quad \Longrightarrow \quad g(<B, \beta_1>) \vdash_{\mathfrak{A}_{\mathrm{SM}}(P)} g(<B, \beta_2>)$$

Now let the condition $<B, \beta_1> \vdash_{\mathfrak{A}_{\mathrm{SM_m}}(P)} <B, \beta_2>$ be true. Again, one can easily prove, that the old machine can do a step $g(<B, \beta_1>) \vdash_{\mathfrak{A}_{\mathrm{SM}}(P)} <A, \alpha_2>$. Now we have to prove that $<A, \alpha_2> = g(<B, \beta_2>)$. The function g essentially applies 'encode', i.e. it is sufficient to prove, that:

$$\mathrm{encode}(\beta_2(RS), \beta_2(CHI), \beta_2(SS)) = \alpha_2(PS)$$

The last Lemma proved that this is equivalent to:

$$\mathrm{decode}(\alpha_2(PS)) = (\beta_2(RS), \beta_2(CHI), \beta_2(SS))$$

Now assume, this condition is violated. With the last theorem, we can conclude that

$$h(g(<B, \beta_1>)) \vdash_{\mathfrak{A}_{\mathrm{SM_m}}(P)} <B, \beta_3>$$

and $<B, \beta_3> \neq h(g(<B, \beta_2>))$. Because of the last Lemma, we also have $h(g(<B, \beta_i>)) = <B, \beta_i>$ for $i = 1, 2$, i.e. we have

$$<B, \beta_1> \begin{array}{ll} \vdash_{\mathfrak{A}_{\mathrm{SM_m}}(P)} & <B, \beta_2> \\ & \neq \\ \vdash_{\mathfrak{A}_{\mathrm{SM_m}}(P)} & <B, \beta_3> \end{array}$$

Now we have a contradiction to the assumption, because $\vdash_{\mathfrak{A}_{\mathrm{SM_m}}(P)}$ is deterministic. q.e.d.

8 Equivalence of $\mathfrak{A}_{\mathrm{SM_m}}(P)$ and $\mathfrak{M}[\![P]\!]$

In this final section, we will prove the main result of this paper, the equivalence of the denotational semantics of PLP and its implementation. Remember from Section 1 that $\mathfrak{M}[\![P]\!]$ is a mapping of type $\mathbb{Z}^n \dashrightarrow \mathbb{Z}^n$ for a program P with n input/output variables. In order to relate this function to the stack machine, we first have to define which function is computed by an (incomplete) execution of a stack machine.

Definition 3 (Computed function) Let $P \in \underline{\texttt{PLP-Prog}}$ and $a_1, a_2 \in \mathbb{N}_0$. We define the **computed function of $\mathfrak{A}_{\mathrm{SM_m}}(P)$ between a_1 and a_2** by:

$$\begin{array}{rcl}
\Phi_{P,a_1,a_2} : \underline{\mathrm{S}} & \dashrightarrow & \underline{\mathrm{S}} \\
\Phi_{P,a_1,a_2}(\chi_1) := \chi_2 & \Longleftrightarrow & \text{there is a minimal } j \in \mathbb{N}_0 \text{ such that} \\
& & \forall <A,\alpha_1>, <A,\alpha_2> \in \underline{\mathrm{states}}_{\mathfrak{A}_{\mathrm{SM_m}}(P)} \text{ with}
\end{array}$$

(i) $<A,\alpha_1>$ reachable,

(ii) $<A,\alpha_1> \vdash^{j}_{\mathfrak{A}_{\mathrm{SM_m}}(P)} <A,\alpha_2>$,

(iii) $\alpha_1(DS) = \alpha_2(DS)$, $\alpha_1(RS) = \alpha_2(RS)$, $\alpha_1(SS) = \alpha_2(SS)$

(iv) $\alpha_i(PC) = a_i$ for $1 \le i \le 2$, and

(v) $\alpha_1(CHI) = \chi_1$

holds, that $\alpha_2(CHI) = \chi_2$.

This means, that we define the associated function by starting the machine at program address a_1 on the variable binding χ_1 and compute until it reaches program address a_2, i.e. before execution the command at this address. The variable binding at this time is the result of the function. $\triangleleft$

The definition is maybe more complicated than expected. Of course, it would be quite easy to define the computed function of a whole program. This definition has the advantage that it enables us to describe the computation of *parts* of the program. By doing so, we are in the position to use the compositional structure of the semantics.

Obviously, the computation of the whole program can be described by $\Phi_{P,1,0}$.

Corollary 5 Let $P \in \underline{\texttt{PLP-Prog}}$, $\Gamma_1 \ldots \Gamma_q = \mathrm{trans_m}[\![P]\!]$, $E \in \underline{\texttt{PLP-Exp}}$, $st \in \underline{\mathrm{Tab}}$, $l \in \mathbb{N}_0$ and $a_1, a_2 \in \{1 \ldots, q\}$ such that $\Gamma_{a_1} \ldots \Gamma_{a_2} = \mathrm{et_m}[\![E]\!](st, l)$. Furthermore, let $<A,\alpha_1>, <A,\alpha_2> \in \underline{\mathrm{states}}_{\mathfrak{A}_{\mathrm{SM_m}}(P)}$ with

(i) $<A,\alpha_1>$ reachable,

(ii) $<A,\alpha_1> \vdash^{(a_2-a_1+1)}_{\mathfrak{A}_{\mathrm{SM_m}}(P)} <A,\alpha_2>$, and

(iii) $\alpha_1(PC) = a_1$ and $\alpha_2(PC) = a_2 + 1$.

Then we have:

$$\alpha_2(DS) = (\mathfrak{C}[\![E]\!](\alpha_1(CHI))) : \alpha_1(DS), \quad \alpha_2(CHI) = \alpha_1(CHI), \quad \alpha_2(PS) = \alpha_1(PS)$$

This means that the execution of the code of the translation of E only pushes the value of E onto the data stack. $\diamondsuit$

Due to the similar method of variable access, the proof is just an easy induction on the structure of E.

Now we will proof that the translation of a command is correct, as long as the procedures, which can be called from within the command, are translated in a correct way.

Theorem 6 Let $P \in \underline{\texttt{PLP-Prog}}$, $\Gamma_1 \ldots \Gamma_q = \mathrm{trans_m}[\![P]\!]$, $\Gamma \in \underline{\texttt{PLP-Cmd}}$, $st \in \underline{\mathrm{Tab}}$, $l \in \mathbb{N}_0$ and $a_1, a_2 \in \{1, \ldots, q\}$ such that $\Gamma_{a_1} \ldots \Gamma_{a_2} = \mathrm{ct_m}[\![\Gamma]\!](st, l, a_1)$. Define the set of all procedures callable with st by

$$P_{st} := \{\texttt{P}_{l_i,o_i} \in \underline{\texttt{PIde}} \mid\mid st(\texttt{P}_{l_i,o_i}) = (ca_{l_i,o_i}, t_{l_i,o_i})\}$$

and let $P_{st} = \{\texttt{P}_{l_1,o_1}, \ldots, \texttt{P}_{l_k,o_k}\}$. Furthermore, assume that there is a function $\Phi_{l_i,o_i} \in \underline{\mathrm{C}}$ for all $1 \leq i \leq k$ with $st(\texttt{P}_{l_i,o_i}) = (ca_{l_i,o_i}, t_{l_i,o_i})$ such that

$$\Gamma_b = \texttt{CALL}(ca_{l_i,o_i}, l_i, t_{l_i,o_i}) \quad \Longrightarrow \quad \Phi_{l_i,o_i} = \Phi_{P,b,b+1} \qquad \forall 1 \leq b \leq q$$

holds. Let $\pi := [\texttt{P}_{l_1,i_1}/\Phi_{l_1,i_1}, \ldots, \texttt{P}_{l_k,i_k}/\Phi_{l_k,i_k}]$. Then the code for Γ is correct:

$$\mathfrak{C}[\![\Gamma]\!](l, \pi) = \Phi_{P,a_1,a_2+1} \qquad \diamondsuit$$

Proof Induction on the structure of Γ:

$\boxed{\Gamma = \texttt{X}_{dl,o} := E\text{:}}$ First of all, we will expand the conclusion (ii):

$$\begin{array}{ll}
& \mathfrak{C}[\![\Gamma]\!](l, \pi) = \Phi_{P,a_1,a_2+1} \\
\Longleftrightarrow & (\ \forall \chi \in \underline{\mathrm{S}} : \mathfrak{C}[\![\Gamma]\!](l, \pi, \chi) = \chi' \iff \Phi_{P,a_1,a_2+1}(\chi) = \chi'\ ) \\
\Longleftrightarrow & (\ \forall \chi \in \underline{\mathrm{S}} : \mathfrak{C}[\![\Gamma]\!](l, \pi, \chi) = \chi' \iff \text{there is a minimal } j \in \mathbb{N}_0 \text{ such that} \\
& \qquad \forall <A, \alpha_1>, <A, \alpha_2> \in \underline{\mathrm{states}}_{\mathfrak{A}_{\mathrm{SM_m}}(P)} \text{ with} \\
& \qquad \text{(i) } <A, \alpha_1> \text{ reachable,} \\
& \qquad \text{(ii) } <A, \alpha_1> \vdash^{j}_{\mathfrak{A}_{\mathrm{SM_m}}(P)} <A, \alpha_2>, \\
& \qquad \text{(iii) } \alpha_1(DS) = \alpha_2(DS),\ \alpha_1(RS) = \alpha_2(RS), \\
& \qquad \qquad \alpha_1(SS) = \alpha_2(SS) \\
& \qquad \text{(iv) } \alpha_1(PC) = a_1,\ \alpha_2(PC) = a_2 + 1, \text{ and} \\
& \qquad \text{(v) } \alpha_1(CHI) = \chi \\
& \qquad \text{holds, that } \alpha_2(CHI) = \chi'.\ )
\end{array}$$

Now let $\chi, <A, \alpha_1>, <A, \alpha_2> \in \underline{\text{states}}_{\mathfrak{A}_{\mathrm{SM_m}}(P)}$ be arbitrary, but fixed with conditions (i)—(v). The semantics of Γ can be expressed by:

$$\mathfrak{C}[\![\mathtt{X}_{dl,o} := E]\!](l, \pi, \chi) = \chi[\mathtt{X}_{dl,o}/\mathfrak{C}[\![E]\!](\chi)], \text{ if } \chi(\mathtt{X}_{dl,o}) \text{ defined}$$

On the other hand, we have:

$$\mathrm{ct_m}[\![\mathtt{X}_{dl,o} := E]\!](st, l, a_1) = \begin{bmatrix}\mathrm{et_m}[\![E]\!](st, l) \\ \mathtt{PUTX}\ (dl, o)\end{bmatrix} \qquad \text{if } st(\mathtt{X}_{dl,o}) = (\mathtt{var})$$

By applying the last corollary to $\mathrm{et_m}[\![E]\!](st, l)$ we get a state $<A, \alpha_3> \in \underline{\text{states}}_{\mathfrak{A}_{\mathrm{SM_m}}(P)}$ with

(i) $<A, \alpha_1> \vdash^{(a_2-1-a_1+1)}_{\mathfrak{A}_{\mathrm{SM_m}}(P)} <A, \alpha_3>$

(ii) $\alpha_1(PC) = a_1$ and $\alpha_3(PC) = a_2$

(iii) $\alpha_3(DS) = \mathfrak{C}[\![E]\!](\alpha_1(CHI)) : \alpha_1(DS)$, $\quad \alpha_3(CHI) = \alpha_1(CHI)$ and $\alpha_3(PS) = \alpha_1(PS)$

Because the machine is deterministic, it is clear that $<A, \alpha_2>$ can only be reached via $<A, \alpha_3>$. Hence, $<A, \alpha_1> \vdash^{(a_2-a_1+1)}_{\mathfrak{A}_{\mathrm{SM_m}}(P)} <A, \alpha_2>$ is the shortest possible computation. The execution of the PUTX command changes CHI and DS in the following way:

$$\begin{aligned} \alpha_2(DS) &= \alpha_1(DS) \\ \alpha_2(CHI) &= \alpha_3(CHI)[\mathtt{X}_{dl,o}/\mathfrak{C}[\![E]\!](\alpha_1(CHI))] \\ &= \alpha_1(CHI)[\mathtt{X}_{dl,o}/\mathfrak{C}[\![E]\!](\alpha_1(CHI))] \end{aligned}$$

With $\chi = \alpha_1(CHI)$ we have:

$$\alpha_2(CHI) = \chi[\mathtt{X}_{dl,o}/\mathfrak{C}[\![E]\!](\chi)] = \mathfrak{C}[\![\mathtt{X}_{dl,o} := E]\!](l, \pi, \chi) = \chi'$$

which completes this case.

$\boxed{\Gamma = \mathtt{P}_{l_i,o_i}():}$ We have $\mathtt{P}_{l_i,o_i} \in \mathtt{P}_{st}$, thus we have a function $\Phi_{l_i,o_i} \in \underline{\mathrm{C}}$ with

$$\mathfrak{C}[\![\mathtt{P}_{l_i,o_i}()]\!](l, \pi, \chi) = (\pi(\mathtt{P}_{l_i,o_i}))(\chi) = \Phi_{l_i,o_i}$$

Hence, the function Φ_{l_i,o_i} fulfills the condition

$$\Gamma_b = \mathtt{CALL}(ca_{l_i,o_i}, l_i, t_{l_i,o_i}) \quad \Longrightarrow \quad \Phi_{l_i,o_i} = \Phi_{P,b,b+1} \qquad \forall 1 \le b \le q \qquad (*)$$

On the other hand, we have

$$\mathrm{ct_m}[\![\mathtt{P}_{l_i,o_i}()]\!](st, l, a_1) = \mathtt{CALL}(ca_{l_i,o_i}, l_i, t_{l_i,o_i}) \quad \text{where } st(\mathtt{P}_{l_i,o_i}) = (ca_{l_i,o_i}, t_{l_i,o_i})$$

We apply $(*)$ to $b = a_1$ and conclude

$$\mathfrak{C}[\![\mathtt{P}_{l_i,o_i}()]\!](l, \pi, \chi) = \Phi_{l_i,o_i} = \Phi_{P,a_1,a_1+1}$$

The remaining cases (the induction step, actually) are just standard implementations, and are left to the reader. q.e.d.

The next aim is to extend this result to the translation of blocks.

Theorem 7 Let $P \in \underline{\texttt{PLP-Prog}}$, $\Gamma_1 \ldots \Gamma_q = \mathrm{trans_m}[\![P]\!]$, $B \in \underline{\texttt{PLP-Block}}$, $st \in \underline{\mathrm{Tab}}$, $l \in \mathbb{N}_0$ and $a_1, a_2 \in \{1, \ldots, q\}$ such that $\Gamma_{a_1} \ldots \Gamma_{a_2} = \mathrm{bt_m}[\![B]\!](st, l, a_1)$. Define the set of all procedures callable with st by

$$P_{st} := \{\mathtt{P}_{l_i,o_i} \in \underline{\texttt{PIde}} \mid\mid st(\mathtt{P}_{l_i,o_i}) = (ca_{l_i,o_i}, t_{l_i,o_i})\}$$

and let $P_{st} = \{\mathtt{P}_{l_1,o_1}, \ldots, \mathtt{P}_{l_k,o_k}\}$. Furthermore, assume that there is a function $\Phi_{l_i,o_i} \in \underline{\mathrm{C}}$ for all $1 \leq i \leq k$ with $st(\mathtt{P}_{l_i,o_i}) = (ca_{l_i,o_i}, t_{l_i,o_i})$ such that

$$\Gamma_b = \mathtt{CALL}(ca_{l_i,o_i}, l_i, t_{l_i,o_i}) \implies \Phi_{l_i,o_i} = \Phi_{P,b,b+1} \qquad \forall 1 \leq b \leq q$$

holds. Let $\pi := [\mathtt{P}_{l_1,i_1}/\Phi_{l_1,i_1}, \ldots, \mathtt{P}_{l_k,i_k}/\Phi_{l_k,i_k}]$ and $c \in \{1, \ldots, q\}$ with $\Gamma_c = \mathtt{CALL}(a_1, l, \mathrm{size}[\![B]\!])$ be a call to the code of B. Then we have:

$$\mathfrak{B}[\![B]\!](l, \pi) = \Phi_{P,c,c+1} \qquad \diamondsuit$$

Proof Let $B = \Delta\Gamma = \Delta_v\Delta_p\Gamma = \mathtt{var}\ \mathtt{X}_{l,1}, \ldots, \mathtt{X}_{l,n}\Delta_p\Gamma$. W.l.o.g. we can assume that $\Delta_v \neq \varepsilon$. With the definition of $\mathfrak{B}$ we have

$$\begin{aligned}
\mathfrak{B}[\![B]\!](l, \pi, \chi) &= \mathrm{reset}(\chi_l, l, \mathfrak{C}[\![\Gamma]\!](l, \mathfrak{D}[\![\Delta]\!](l, \pi, \chi))) \\
&= \mathrm{reset}(\chi_l, l, \mathfrak{C}[\![\Gamma]\!](l, \mathfrak{D}[\![\Delta_p]\!](l, \mathfrak{D}[\![\Delta_v]\!](l, \pi, \chi)))) \\
&= \mathrm{reset}(\chi_l, l, \mathfrak{C}[\![\Gamma]\!](l, \mathfrak{D}[\![\Delta_p]\!](l, \pi, \chi[\mathtt{X}_{l,1}/0, \ldots, \mathtt{X}_{l,n}/0])))
\end{aligned}$$

for all $\chi \in \underline{\mathrm{S}}$.
In the sequel we will fix $\chi \in \underline{\mathrm{S}}$. With the definition of the computed function we have the equivalence

$\mathfrak{B}[\![B]\!](l, \pi, \chi) = \Phi_{P,c,c+1}(\chi) \iff$ there is a minimal $j \in \mathbb{N}_0$ such that $\forall$ $<A, \alpha_1>, <A, \alpha_2> \in \underline{\mathrm{states}}_{\mathfrak{A}_{\mathrm{SM_m}}(P)}$ with

(i) $<A, \alpha_1>$ reachable,

(ii) $<A, \alpha_1> \vdash^{j}_{\mathfrak{A}_{\mathrm{SM_m}}(P)} <A, \alpha_2>$,

(iii) $\alpha_1(DS) = \alpha_2(DS)$, $\alpha_1(RS) = \alpha_2(RS)$, $\alpha_1(SS) = \alpha_2(SS)$

(iv) $\alpha_1(PC) = c$ and $\alpha_2(PC) = c + 1$, and

(v) $\alpha_1(CHI) = \chi$

holds, that $\alpha_2(CHI) = \mathfrak{B}[\![B]\!](l, \pi, \chi)$.

Now let $\chi, <A, \alpha_1>, <A, \alpha_2> \in \underline{\mathrm{states}}_{\mathfrak{A}_{\mathrm{SM_m}}(P)}$ be arbitrary, but fixed with conditions (i)—(v).
The modified machine executes the command Γ_c and enters the uniquely defined intermediate state $<A, \alpha_3> \in \underline{\mathrm{states}}_{\mathfrak{A}_{\mathrm{SM_m}}(P)}$ with

(i) $<A,\alpha_1>\vdash_{\mathfrak{A}_{\mathrm{SM_m}}(P)}<A,\alpha_3>$,

(ii) $\alpha_3(PC)=a_1$,

(iii) $\alpha_3(RS)=c+1:\alpha_1(RS)$,

(iv) $\alpha_3(SS)=(l,n,\alpha_1(\mathit{CHI})_l):\alpha_1(SS)=(l,n,\chi_l):\alpha_1(SS)$, and

(v) $\alpha_3(\mathit{CHI})=\alpha_1(\mathit{CHI})[\mathtt{X}_{l,1}/0,\ldots,\mathtt{X}_{l,n}/0]=\chi[\mathtt{X}_{l,1}/0,\ldots,\mathtt{X}_{l,n}/0]$

The translation of B looks like this:

$$\Gamma_{a_1}\ldots\Gamma_{a_2}=\mathrm{bt_m}[\![\Delta\Gamma]\!](st,l,a_1)=\begin{bmatrix}\mathrm{ct_m}[\![\Gamma]\!](\mathrm{upd}[\![\Delta]\!](st,a_1+\mathrm{len}[\![\Gamma]\!]+1),l,a_1)\\ \mathtt{RET}\\ \mathrm{dt_m}[\![\Delta]\!](\mathrm{upd}[\![\Delta]\!](st,a_1+\mathrm{len}[\![\Gamma]\!]+1),l,a_1+\mathrm{len}[\![\Gamma]\!]+1)\end{bmatrix}$$

and the effect of the variable declaration on the symbol table is

$$\begin{aligned}\mathrm{upd}[\![\Delta]\!](st,a_1+\mathrm{len}[\![\Gamma]\!]+1) &= \mathrm{upd}[\![\Delta_p]\!](\mathrm{upd}[\![\Delta_v]\!](st,a_1+\mathrm{len}[\![\Gamma]\!]+1),a_1+\mathrm{len}[\![\Gamma]\!]+1)\\ &= \mathrm{upd}[\![\Delta_p]\!](st[\mathtt{X}_{l,1}/(\mathtt{var}),\ldots,\mathtt{X}_{l,n}/(\mathtt{var})],a_1+\mathrm{len}[\![\Gamma]\!]+1)\end{aligned}$$

Now we will proof the theorem by induction on the nesting level h of B, i.e. on the number of times B can be decomposed into (smaller) blocks.

$\boxed{h=0:}$ This means B cannot be decomposed, i.e. $\Delta_p=\varepsilon$. As a consequence the effect of Δ on the symbol table is only

$$\mathrm{upd}[\![\Delta]\!](st,a_1+\mathrm{len}[\![\Gamma]\!]+1)=st[\mathtt{X}_{l,1}/(\mathtt{var}),\ldots,\mathtt{X}_{l,n}/(\mathtt{var})]$$

and the code produced by the declaration translation is empty:

$$\mathrm{dt_m}[\![\Delta]\!](\mathrm{upd}[\![\Delta]\!](st,a_1+\mathrm{len}[\![\Gamma]\!]+1),l,a_1+\mathrm{len}[\![\Gamma]\!]+1)=\varepsilon$$

Now we can apply the last theorem and we obtain

$$\mathfrak{C}[\![\Gamma]\!](l,\pi,\alpha_3(\mathit{CHI}))=\Phi_{P,a_1,a_2}(\alpha_3(\mathit{CHI}))$$

This means we get another intermediate state $<A,\alpha_4>\in\underline{\mathrm{states}}_{\mathfrak{A}_{\mathrm{SM_m}}(P)}$ with

(i) $<A,\alpha_3>\vdash^{j'}_{\mathfrak{A}_{\mathrm{SM_m}}(P)}<A,\alpha_4>$,

(ii) $\alpha_4(PC)=a_2$,

(iii) $\alpha_3(DS)=\alpha_4(DS)$, $\alpha_3(RS)=\alpha_4(RS)$, $\alpha_3(SS)=\alpha_4(SS)$,

(iv) $\alpha_4(\mathit{CHI})=\mathfrak{C}[\![\Gamma]\!](l,\pi,\alpha_3(\mathit{CHI}))$.

The next command to be executed is `RET`, which transfers the machine into the state $<A,\alpha_2>$

$$\alpha_2(PC)=c+1,\quad \alpha_2(RS)=\alpha_1(RS),\quad \alpha_2(SS)=\alpha_1(SS)$$

and

$$\begin{aligned}\mathfrak{B}[\![B]\!](l,\pi,\chi) &= \mathrm{reset}(\chi_l, l, \mathfrak{C}[\![\Gamma]\!](l,\pi,\chi[\mathtt{X}_{l,1}/0,\ldots,\mathtt{X}_{l,n}/0]))\\ &= \mathrm{reset}(\chi_l, l, \alpha_4(CHI))\\ &= \mathrm{reset}(\alpha_1(CHI)_l, l, \alpha_4(CHI))\\ &= \alpha_2(CHI)\end{aligned}$$

$\boxed{0,\ldots,h \to h+1:}$ Let the nesting level of B be equal to $h+1$. We assume the theorem to be proved for all blocks with nesting level less than $h+1$. Now we have $\Delta_p = \mathtt{proc}\ \mathtt{P}_{l,1}; B_1; \ldots; \mathtt{P}_{l,m}; B_m$. This means that the effect of Δ on the symbol table is

$$\begin{aligned} st' := \mathrm{upd}[\![\Delta]\!](st, a_1+\mathrm{len}[\![\Gamma]\!]+1) &= \mathrm{upd}[\![\Delta_p]\!](st[\mathtt{X}_{l,1}/(\mathtt{var}),\ldots,\mathtt{X}_{l,n}/(\mathtt{var})], a_1+\mathrm{len}[\![\Gamma]\!]+1)\\ &= st[\mathtt{X}_{l,1}/(\mathtt{var}),\ldots,\mathtt{X}_{l,n}/(\mathtt{var}),\\ &\quad \mathtt{P}_{l,1}/(\mathtt{proc}, a_1+\mathrm{len}[\![\Gamma]\!]+1, \mathrm{size}[\![B_1]\!]),\\ &\quad \mathtt{P}_{l,2}/(\mathtt{proc}, a_1+\mathrm{len}[\![\Gamma]\!]+1+\mathrm{len}[\![B_1]\!], \mathrm{size}[\![B_2]\!]),\\ &\quad \vdots\\ &\quad \mathtt{P}_{l,m}/(\mathtt{proc}, a_1+\mathrm{len}[\![\Gamma]\!]+1+\sum_{k=1}^{m-1}\mathrm{len}[\![B_k]\!], \mathrm{size}[\![B_m]\!])]\end{aligned}$$

and the code produced by the declaration translation is

$$\mathrm{dt_m}[\![\Delta]\!](st', l, a_1+\mathrm{len}[\![\Gamma]\!]+1) = \begin{bmatrix}\mathrm{bt}[\![B_1]\!](st', l+1, a_1+\mathrm{len}[\![\Gamma]\!]+1)\\ \mathrm{bt}[\![B_2]\!](st', l+1, a_1+\mathrm{len}[\![\Gamma]\!]+1+\mathrm{len}[\![B_1]\!])\\ \vdots\\ \mathrm{bt}[\![B_m]\!](st', l+1, a_1+\mathrm{len}[\![\Gamma]\!]+1+\sum_{k=1}^{m-1}\mathrm{len}[\![B_k]\!])\end{bmatrix}$$

On the other hand, the declaration semantics used in the block semantics is

$$\mathfrak{D}[\![\Delta_p]\!](l,\pi,\chi[\mathtt{X}_{l,1}/0,\ldots,\mathtt{X}_{l,n}/0]) = (\pi[\mathtt{P}_{l,1}/\Phi_1,\ldots,\mathtt{P}_{l,m}/\Phi_m], \chi[\mathtt{X}_{l,1}/0,\ldots,\mathtt{X}_{l,n}/0])$$

where the functions $\Phi_1,\ldots,\Phi_m$ are the least fixpoint of $\mathcal{S}_\Delta : \underline{C}^m \dashrightarrow \underline{C}^m$, defined by

$$(\mathcal{S}_\Delta(\Phi'_1,\ldots,\Phi'_m))_i := \mathfrak{B}[\![B_i]\!](l+1, \pi[\mathtt{P}_{l,1}/\Phi'_1,\ldots,\mathtt{P}_{l,m}/\Phi'_m])$$

Of course, we want to use the last theorem again. But in order to do so, we have to ensure, that the code for each block B_i corresponds to its semantics, i.e. we have to prove that for all $1 \le i \le m$:

$$\Gamma_{b_i} = \mathtt{CALL}(a_1+\mathrm{len}[\![\Gamma]\!]+1+\sum_{k=1}^{i-1}\mathrm{len}[\![B_i]\!], l+1, \mathrm{size}[\![B_i]\!]) \implies \Phi_i = \Phi_{P,b_i,b_i+1} \qquad \forall 1 \le b_i \le q$$

In this case, we can finish the proof in the same way as in the case $h = 0$.
For the rest of the proof we will fix $b_1,\ldots,b_m \in \{1,\ldots,q\}$ with the above condition. We have to show, that

$$(\Phi_1,\ldots,\Phi_m) = (\Phi_{P,b_1,b_1+1},\ldots,\Phi_{P,b_m,b_m+1})$$

Because $(\Phi_1,\ldots,\Phi_m)$ is the least fixpoint of $\mathcal{S}_\Delta$ we can divide this into two parts:

(i) we show that $(\Phi_{P,b_1,b_1+1},\ldots,\Phi_{P,b_m,b_m+1})$ is a fixpoint of $\mathcal{S}_\Delta$, too, i.e.:

$$\mathcal{S}_\Delta\ (\Phi_{P,b_1,b_1+1},\ldots,\Phi_{P,b_m,b_m+1}) = \Phi_{P,b_1,b_1+1},\ldots,\Phi_{P,b_m,b_m+1} \qquad (*)$$

Now all blocks have a nesting level less or equal h, i.e. we can use the theorem with the bindings $[\mathtt{P}_{l,i}/\Phi_{P,b_i,b_i+1} \mid\mid 1 \leq i \leq m]$ and get

$$\mathfrak{B}[\![B_i]\!](l+1, \pi[\mathtt{P}_{l,1}/\Phi_{P,b_1,b_1+1},\ldots,\mathtt{P}_{l,m}/\Phi_{P,b_m,b_m+1}]) \quad \forall 1 \leq i \leq m$$

which is equivalent to $(*)$ with the definition of $\mathcal{S}_\Delta$.

(ii) we show that $\Phi_{P,b_1,b_1+1},\ldots,\Phi_{P,b_m,b_m+1}$ is also a least fixpoint, i.e.:

$$(\Phi_{P,b_1,b_1+1},\ldots,\Phi_{P,b_m,b_m+1}) \sqsubseteq (\Phi_1,\ldots,\Phi_m)$$

Suppose this is not fulfilled, i.e. there is an $i \in \{1,\ldots,m\}$ such that $\Phi_{P,b_i,b_i+1} \sqsupset \Phi_i$. This means there are $\chi,\chi' \in \underline{\mathrm{S}}$ with

$$\Phi_{P,b_i,b_i+1}(\chi) = \chi' \quad \text{and} \quad \Phi_i(\chi) \text{ undefined}$$

There are only two possibilities for $\Phi_i(\chi)$ to become undefined:

(a) The corresponding block B_i contains an assignment $\mathtt{X}_{o,p} := E$ where the variable $\mathtt{X}_{o,p}$ is not defined and during evaluation of $\Phi_i(\chi)$ this assignment is evaluated. In this case we have a corresponding $\mathtt{PUTX}(o,p)$ in the stack code for B_i and the command will fail, because the variable $\mathtt{X}_{o,p}$ is undefined.

(b) B_i contains a call to the precedure $\mathtt{P}_{o,p}$. The corresponding entry in the environment $\pi[\mathtt{P}_{l,1}/\Phi_1,\ldots,\mathtt{P}_{l,m}/\Phi_m]$ is undefined for the variable binding at the time of evaluation. Now this $\mathtt{P}_{o,p}$ must be in the set $\{\mathtt{P}_{l,1},\ldots,\mathtt{P}_{l,m}\}$. We can iterate this argument and get an infinite sequence of evaluations of procedures $\mathtt{P}_{l,i}$. Each step corresponds to a procedure call in the stack machine. This means there is an infinite sequence of procedure calls in the stack machine an this is a contradiction to $\Phi_{P,b_i,b_i+1}(\chi) = \chi'$. q.e.d.

At least, we are able to state the equivalence of $\mathfrak{A}_{\mathrm{SM_m}}(P)$ and $\mathfrak{M}[\![P]\!]$.

Corollary 6 For each $P \in \underline{\mathtt{PLP\text{-}Prog}}$ and $z_1,\ldots,z_m \in \mathbb{Z}$ holds:

$$\mathfrak{M}[\![P]\!](z_1,\ldots,z_m) = (z'_1,\ldots,z'_m) \iff \Phi_{P,1,0}([\mathtt{X}_{0,1}/z_1,\ldots,\mathtt{X}_{0,m}/z_m]) = [\mathtt{X}_{0,1}/z'_1,\ldots,\mathtt{X}_{0,m}/z'_m]$$

$\diamondsuit$

Proof Let $P = \mathtt{in/outX}_{0,1},\ldots,\mathtt{X}_{0,n};B.$. We distinguish two cases:

(i) $n \neq m$

In this case $\mathfrak{M}[\![P]\!](z_1,\ldots,z_m)$ is not defined. Now assume, that there is a reachable state $< A,\alpha_1 > \in \underline{\mathrm{states}}_{\mathfrak{A}_{\mathrm{SM_m}}(P)}$ with $\alpha_1(CHI) = [\mathtt{X}_{0,1}/z_1,\ldots,\mathtt{X}_{0,m}/z_m]$ and $\alpha_1(PC) = 1$. Of course, this must be an initial state. But then we have

$$\begin{aligned} & \alpha_1(\mathit{defnd})(\alpha_1(CHI),0) = \alpha_1(\mathit{arity})) \\ \iff & \alpha_1(\mathit{defnd})([\mathtt{X}_{0,1}/z_1,\ldots,\mathtt{X}_{0,m}/z_m],0) = n \\ \iff & m = n \qquad \lightning \end{aligned}$$

Now we know that there is no such state and as a consequence $\Phi_{P,1,0}$ is undefined.

(ii) $n = m$

Now we have

$$\mathfrak{M}[\![\texttt{in/out X}_{0,1},\ldots,\texttt{X}_{0,n}; B.]\!](z_1,\ldots,z_n) \;:=\; (\chi(\texttt{X}_{0,1}),\ldots,\chi(\texttt{X}_{0,n})),$$
$$\text{where } \chi = \mathfrak{B}[\![B]\!](1,[],[\texttt{X}_{0,1}/z_1,\ldots,\texttt{X}_{0,n}/z_n])$$

and on the other hand:

$$\text{trans}_m[\![\texttt{in/out X}_{0,1},\ldots,\texttt{X}_{0,n}; B.]\!] := \begin{bmatrix} \texttt{CALL }(3,0,\text{size}[\![B]\!]) \\ \texttt{JMP 0} \\ \text{bt}_m[\![B]\!]([(\texttt{X}_{0,1}/(\texttt{var}),\ldots,(\texttt{X}_{0,n}/(\texttt{var})],3,1) \end{bmatrix}$$

Now we can apply the last theorem (note that π=[]) and we get

$$\begin{aligned}\mathfrak{B}[\![B]\!](1,[],[\texttt{X}_{0,1}/z_1,\ldots,\texttt{X}_{0,n}/z_n]) &= \Phi_{P,1,2}([\texttt{X}_{0,1}/z_1,\ldots,\texttt{X}_{0,n}/z_n]) \\ &= \Phi_{P,1,0}([\texttt{X}_{0,1}/z_1,\ldots,\texttt{X}_{0,n}/z_n])\end{aligned}$$

And this completes this case. q.e.d.

Conclusions

We have presented a proof for the standard implementation of imperative programming languages with nested procedure declaration and recursion. The description of the stack machine was given in terms of an evolving algebra, which turned out to be appropriate for this task. The description is without notational overhead, it is easy to read and the reader can understand it, even without exactly knowing the definition of an evolving algebra.

The main problem was to cope with the run-time structure of the procedure stack. The notion of admissibility we have introduced reflects exactly the intention of the procedure stack as a sequence of activation records, which are linked in a way, which implements the scoping rules of the programming language.

Based on this knowledge, we were able to separate the three different kinds of information stored in the procedure stack. The resulting modified machine was proofed to be an equivalent evolving algebra. This was straightforward, because the machines perform essentially the same steps and just store the information in a different way. A little drawback was the necessity to introduce a component in each entry of the save stack, which is not needed by the machine at all. The notion of equivalence connected with evolving algebras forced us to make the states of the machines to be convertible. The problem is to reconstruct the state of the original machine from the state of the modified machine. For this direction, we needed the extra information stored in the components of the save stack. It may be interesting to investigate,whether there is an alternative notion of equivalence of evolving algebras without this flaw.

The modified machine is on a higher level of abstraction, i.e. it is closer related to the denotational semantics. This feature made it easy to prove the equivalence. In this context, we saw the necessity to relate evolving algebras to the 'outside world'. At this point we needed an extra definition for the computed function of an evolving algebra, which describes another essential point of abstract machines.

Appendix: Notations

- $\mathbb{N} = \{1, 2, 3, \ldots\}$ the set of natural numbers.
- $\mathbb{N}_0 = \{0, 1, 2. \ldots\}$ the set of natural numbers including 0.
- $f : A \dashrightarrow B$ denotes a partial function from set A to set B.
- $\mathrm{Dom}(f) = \{a \in A \mid\mid f(a) \text{ defined}\}$ the domain of the partial function $f : A \dashrightarrow B$.
- $f \sqsubseteq g$, where $f, g : A \dashrightarrow B$, means $f(a) = b \Rightarrow g(a) = b$, i.e. f is at most as defined as g.
  The set of partial functions from A to B together with this ordering form a complete partial order (cpo).
- $\bigsqcup F$ is the least upper bound of chain F which is a subset of a cpo.
- $f[a/b]$ denotes the modification of f at the position a to the value b, i.e. if $f : A \dashrightarrow B$, then $f[a/b] : A \dashrightarrow B$ is defined by:

$$(f[a/b])(x) := \begin{cases} b, & \text{if } x = a; \\ f(x), & \text{otherwise.} \end{cases}$$

- $(\ldots)_i$ selects the i-th component of its argument, i.e. if $a_1, \ldots, a_n \in A$ then $(a_1, \ldots, a_n)_i = a_i$.
- $p.n$ with $p \in \mathbb{Z}^\star$ and $n \in \mathbb{N}_0$ selects the n-th character of the word p, i.e. if $p = z_1 \ldots z_m$ then $p.n = z_n$
- We don't use λ-expressions to denote construction of new functions from old ones, but partial application instead, i.e. if $f : A \times B \dashrightarrow C$, $a \in A$, then $f(a) : B \dashrightarrow C$. This corresponds to the λ-expression $\lambda b.f(a, b)$.

References

[BR92] E. Börger and D. Rosenzweig. The WAM — Definition and Compiler Correctness. Technical Report TR–14/92, Dipartimento di Informatica, Universita di Pisa, 1992.

[GM87] Y. Gurevich and J. M. Morris. Algebraic Operational Semantics and Modula-2*. In M. M. Richter E. Börger, H. Büning, editor, *Proceedings of the 4th Workshop on Computer Science Logic*, number 329 in Lecture Notes in Computer Science, pages 83–101, 1987.

[Gur88] Y. Gurevich. Logic and the Challenge of Computer Science. In E. Börger, editor, *Trends in theoretical computer science*, pages 1–57. Computer Science Press, 1988.

[Gur91] Y. Gurevich. Evolving Algebras: A Tutorial Introduction. *Bulletin of the European Association for Theoretical Computer Science*, 43:264–284, February 1991.

[HJ70] W. Henhapl and C. B. Jones. The block concept and some possible implementations, with proofs of equivalence. Technical Report TR 25.104, IBM Laboratory Vienna, April 1970.

[Ind88] K. Indermark. Functional Compiler Description. *Mathematical Problems in Computation Theory*, 21:257–275, 1988.

[JL71] C. B. Jones and P. Lucas. Proving correctness of implementation techniques. In B. Eckmann A. Dold, editor, *Symposium on Semantics of Algorithmic Languages*, volume 188 of *Lecture Notes in Mathematics*, pages 178—211. Springer-Verlag, 1971.

[MS76] R. Milne and C. Strachey. *A theory of programming language semantics.* Chapman and Hall, 1976.

[Pol81] W. Polak. *Compiler Specification and Verification*, volume 124 of *Lecture Notes in Compter Science.* Springer–Verlag, 1981.

[Wec92] W. Wechler. *Universal Algebra for Computer Scientists*, volume 25 of *EATCS Monographs on Theoretical Computer Science.* Springer–Verlag Berlin Heidelberg, 1992.