Linear Projective Program Syntax

J.A. Bergstra[1,2] and I. Bethke[2]

[1]Applied Logic Group, Department of Philosophy, Utrecht University
[2]Programming Research Group, Faculty of Science, University of Amsterdam

Abstract. Based on an extremely simple program notation more advanced program features can be developed in linear projective program syntax such as conditional statements, while loops, recursion, use of an evaluation stack, object classes, method calls etc. Taking care of a cumulative and bottom up introduction of such complex features while providing appropriate projections into the lower levels of language development keeps all definitions rigorous and ensures a clear meaning of higher program constructs.

Keywords: Object oriented programming; Program algebra; Program syntax; Program semantics

1. Introduction

The study of programs as text has a profound history in the setting of formal languages and grammars. The path from program texts to program execution has been subject to many investigations and to prove correctness of implementations sophisticated tools have been used in the settings of operational, denotational and axiomatic frameworks. Of current interest are structural operational semantics [AFV01], natural and reduction semantics [Kah87], [FeF87], abstract state machines [Gur95], Scott-Strachey and monadic semantics [Mos90],[Mog91], predicate transformer semantics [Dij75], Hoare [Hoa69] and dynamic logic [Har84], and many more. These different semantic frameworks have been used during the design of programming languages such as Facile, Ada, ML, Prolog, Occam, and Pascal. More recently, operational semantics have been used to provide specifications of Java and the JVM [SSB01], and of the JCVM [Sev04].

This paper is also an exercise in program syntax development. However, in contrast to other approaches, linear projective program syntax (LPPS) aims for practicality and is simple enough to be taught at undergraduate level. Here the emphasis is on *linear projective* syntax that views a program as a sequence of instructions and admits a definition of its semantics by means of a projection into a base language.

The operational semantics of base programs is considered to be a straightforward matter. For each new notation a projection is outlined, where a projection is a transformation of the program notation to a simpler one that has been defined and equipped with a suitable (projective) semantics before. By chaining appropriate projections each program notation can be translated to a base program. The projection thus obtained represents the semantics of the original program. Different projection strategies may project the same program to different pieces of base code. Using polarized process semantics [BeL02],[BeB03] in many

Correspondence and offprint requests to: I. Bethke, Programming Research Group, Faculty of Science, University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands. e-mail: inge@science.uva.nl

cases the semantic equivalence of the different base codes can be established. These matters will not be spelled out in any detail here.

This paper is about definitions only. The absence of theoretical and proven results might be considered a weakness of the approach. It is our view that this is not the case because the primary aim is to obtain easily understood program notations with a rigorous semantic backing based on an absolute minimum of formal meta-theory. It is claimed that no understanding of any proven theorems or of any involved mathematics or logic of any kind is needed (or even useful) for obtaining a semantic understanding of significant parts of imperative programming languages if the line of thought of linear projective semantics is followed.

The paper introduces a number of new techniques from the perspective of program algebra (PGA) [BeL00], [BeL02]: returning jump instructions with parameters improving the expressive power of recursion primitives from [BeB02]. The use of an integrated evaluation and instruction counter stack with object heap improves the model for recursive object oriented programming. The distinction of low level instructions, intermediate language level instructions and high-level instructions admits a flexible structure for the design of projections.

2. The base language

The base language is the LPPS answer to the conceptual question: 'what is a program?'. The tricky aspect is that at the same time as the base language is claimed to provide this answer LPPS predicts that this answer may be in need of refactoring at some future stage. LPPS runs on the unproven assumption that whatever formal definition of a program is given, there will always be a setting in which this definition needs to be compromised or amended in order to get a comprehensible theory. But at the same time an LPPS answer to any problem needs full precision definitions. There is no room for an open ended concept of program like 'a text meant to tell a machine what to do', or—in the absence of precise definitions of 'unit' and of 'deployment'—'a definition of a software component as a unit of deployment'. Likewise, a machine language may not be defined as a program notation meant for machine execution—in the absence of a clear model of the concept of machine execution—and a scripting language is not definable as a language for writing scripts—unless, again the concept of a script is rigorously defined beforehand.

2.1. LPNA, Linear Program Notation with Absolute jumps

As a start for syntax development LPNA may be taken which features among the sequence of the program notations that have been designed on the basis of PGA under the name PGLD [BeL02]. The basis of the definition of LPNA is formed by the notion of a *basic action*. Basic actions may be used as instructions in LPNA. A basic instruction a represents a command issued by the program during execution to its context. That context may involve classical ingredients such as a printing device but it may also involve the operation of a value contained in a variable which is often taken simply as something inside a program. The execution of a basic action a takes place during program execution. It is done by the machine architecture that executes the program and it will have two effects: a possible state change in the execution context and a boolean value which is produced as a result of processing the action—viewed as a request to its environment—and which is subsequently returned to the program under execution where it may be used to decide which is the next instruction to be carried out.

For each basic action a the following are LPNA instructions: a, $+a$ and $-a$. These three versions of the basic actions are called: *void basic action* (a), *positive test instruction* ($+a$), and *negative test instruction* ($-a$). A void basic action represents the command to execute the basic action while ignoring the boolean return value that it produces. Upon completion of the execution of the basic action by the execution environment of the program processing will proceed with the next instruction. The test actions represent the two ways that the boolean reply can be used to influence the subsequent execution of a program. The only other instructions in LPNA are *absolute* jumps $\#\#k$ with k a natural number.

An LPNA program is a finite sequence of LPNA instructions. Execution by default starts at the first (left most) instruction. It ends if the last instruction is executed or if a jump is made to a nonexisting instruction. In particular, $\#\#0$—which will be abbreviated by !—represents program termination because the instruction count is taken to start with 1. The working of test instructions is as follows: if the i-th instruction is $+a$ its execution begins with the execution of a. After completion of that action a reply value is at hand. If

the reply was `true` the run of the program proceeds with instruction number $i+1$—if that exists otherwise execution terminates. Alternatively, if the reply was `false` the execution proceeds with instruction number $i+2$—again assuming its existence, leading to termination in the other case as well. Thus a negative reply enacts that an instruction is skipped. In the case of a negative test instruction $-a$ execution proceeds with instruction number $i+2$ at a positive reply and with instruction number $i+1$ at reply `false`. The exe cution of a jump instruction $\#\#k$ makes the execution continue with the k-th instruction if that exists. Otherwise the program execution terminates. If $\#\#k$ is itself the k-th instruction the execution is caught in a never ending loop.

Example 2.1. Typical LPNA programs are

- $a; +b; !; c; \#\#2$
  which may be read as first do a and then b and then repeat $c; b$ as long as the last reply for b was negative, and terminate after the first positive reply to an execution for b is observed.
- $-a; \#\#6; b; c; !; e; f$
  which may be read as first do a, and if a returns `true` then do b and c, else do e and f, and then terminate.

The definition of LPNA in LPPS has the following intention. At the initial stage of LPPS development it is reasonable to view LPNA as a definition of the concept of a program. Moreover, any text P is a program provided there is at hand a projection ϕ which maps the text to an LP NA instruction sequence $\phi(P)$ which, by definition, represents the meaning of P as a program. The position regarding LPNA is therefore not the untenable assertion that every program is identical to an LPNA instruction sequence but the much more flexible assertion that for some entity P to qualify as a program it must be known how it represents an LPNA instruction sequence. By means of the application of the projection that LPNA program is found and the meaning of the entity P as a program is determined.

2.2. LPNR, Linear Program Notation with Relative jumps

An alternative to LPNA is the program notation LPNR which admits basic actions, positive and negative test actions and the termination instruction !. The jumps are different, however. Only forward jumps ($\#k$) and backward jumps ($\backslash\#k$) (with k a natural number) are admitted. Thus LPNR is LPN with relative jumps. Termination also occurs if a jump is performed outside the range of instructions.[1]

Example 2.2. A typical LPNR program is $+a; \#3; b; !; c; \backslash\#4$ which may be read as first do a and, if the reply to a was negative, continue with b and terminate; otherwise enter an infinite repetition of c.

LPNA and LPNR are intertranslatable by the mappings $\psi : \mathrm{LPNA} \to \mathrm{LPNR}$ and $\phi : \mathrm{LPNR} \to \mathrm{LPNA}$ which transform jumps depending on the place of their occurrence and leave the other instructions unmodified. The mappings are defined by

- $\psi(u_1; \ldots; u_n) = \psi_1(u_1); \ldots; \psi_n(u_n)$ where
$$\psi_i(\#\#k) = \begin{cases} \#k - i & \text{if } k \geq i, \\ \backslash\#i - k & \text{otherwise.} \end{cases}$$
- $\phi(u_1; \ldots; u_n) = \phi_1(u_1); \ldots; \phi_n(u_n)$ where $\phi_i(\#k) = \#\#k + i$ and
$$\phi_i(\backslash\#k) = \begin{cases} \#\#i - k & \text{if } k < i, \\ ! & \text{otherwise.} \end{cases}$$

Example 2.3. The translations of the programs in Example 2.1 and 2.2 are

$$\begin{aligned} \psi(a; +b; !; c; \#\#2) &= \psi_1(a); \psi_2(+b); \psi_3(!); \psi_4(c); \psi_5(\#\#2) \\ &= a; +b; !; c; c; \backslash\#3, \\ \psi(-a; \#\#6; b; c; !; e; f) &= \psi_1(-a); \psi_2(\#\#6); \psi_3(b); \psi_4(c); \psi_5(!); \psi_6(e); \psi_7(f) \\ &= -a; \#4; b; c; !; e; f, \\ \phi(+a; \#3; b; !; c; \backslash\#4) &= \phi_1(+a); \phi_2(\#3); \phi_3(b); \phi_4(!); \phi_5(c); \phi_6(\backslash\#4) \\ &= +a; \#\#5; b; !; c; \#\#2. \end{aligned}$$

[1] The program notation LPNR was originally defined in the setting of PGA in [BeL02] under the name PGLC.

A virtue of LPNR is that it supports *relocatable* programming. Programs may be concatenated without disturbing the meaning of jumps. Placing a program behind another program may be viewed as a relocation which places each instruction at an incremented position. This property can be best exploited if programs are used that always use ! for termination. In some cases it is useful to use only a forward jump to the first missing instr uction for termination. For programs of that form concatenation corresponds to sequential composition.

Another alternative to LPNA as a base language would be LPNAR, the linear program notation with absolute *and* relative jumps. Throughout this paper, however, we shall take LPNA as base language, and projections will be (chained) mappings to LPNA programs. We shall write A $\subseteq$ B for a program notation B extending A and use the operator ϕ for projections. In most cases a projection transforms some program notation back to a simpler one. Sometimes the choice of the right order of such steps is important. A formalism for specifying a strategy for chaining projections will not be included below, however. It is assumed that a reader will be easily able to determine the right order of the various projections. A full precision story may need an annotation of all projection functions with domains and codomains. But as it turns out these matters are often clear from the context and mere ly needed for formalization rather than for rigorous explanation. Therefore that kind of bookkeeping has been omitted, with the implicit understanding that it can always be introduced if additional clarification is necessary.

3. Extensions of LPNA with conditional constructs, while loops, labels and goto's

When transforming a program it may be necessary to insert one or more instructions. Unfortunately that renders the counting of jumps useless. Of course jump counters may be updated simultaneously thus compensating for the introduction of additional instructions. Doing so has proved to lead to unreadable descriptions of projection functions, however, and the device of two level instruction counting will be proposed as a more readable alternative.

Instructions will be split in *first level* instructions and *second level* instructions. The idea is that expanding projections—projections that replace instructions by non-unit instructions—are given in such way that each instruction sequence replacing a single instruction begins with a first level instruction while all subsequent instructions are taken at the second level.

Second level instructions are instructions prefixed with a $\sim$. First level instructions do not have a prefix; they are made second level by prefixing them with a $\sim$. First and second level absolute jumps $\#\#k$ and $\sim \#\#k$ will represent a jump to the k-th first level instruction if that exist (and termination otherwise) simply ignoring the second level instructions. The extension of a LPNA based program notation A with second level instructions is denoted with A:SL. For any A $\supseteq$ LPNA, the projection ϕ : A:SL $\to$ A removes second level instruction markers and updates the jumps.

Projection 3.1. Let LPNA $\subseteq$ A.
ϕ : A:SL $\to$ A is defined by $\phi(u_1;\ldots;u_n) = \phi(u_1);\ldots;\phi(u_n)$ where

$$\phi(\#\#k) = \begin{cases} ! & \text{if } k > max, \\ \#\#k + l & \text{otherwise.} \end{cases}$$

Here max is the number of first level instructions occurring in $u_1;\ldots;u_n$ and l is the number of second level instructions preceding the k-th first level instruction. The other first level instructions remain unmodified. Moreover, $\phi(\sim \#\#k) = \phi(\#\#k)$ and $\phi(\sim u) = u$ for all other first level instructions u.

Example 3.1. $\phi(\sim a; \sim +b; \sim \#\#2; \#\#1) = a; +b; !; \#\#4$

The only use of A:SL is as a target notation for projections from an LPNA based program notation. There will be a number of examples of this in the sequel.

Notation 3.1. For notational convenience we introduce the following abbreviations. We write

- $a\#\#k$ for $a; \sim \#\#k$, $+a\#\#k$ for $+a; \sim \#\#k$, and $-a\#\#k$ for $-a; \sim \#\#k$, and likewise
- $\sim a\#\#k$ for $\sim a; \sim \#\#k$,$\sim +a\#\#k$ for $\sim +a; \sim \#\#k$, $\sim -a\#\#k$ for $\sim -a; \sim \#\#k$.

3.1. Linear projective syntax for the conditional construct

Conditional constructs can be added to LPNA by using four new forms of instruction: for each basic action a, $+a\{$ and $-a\{$ are *conditional header* instructions, further $\}\{$ is the *conditional separator* instruction and $\}$ is the *end of construct* instruction. The idea is that in $+a\{; X; \}\{; Y; \}$ after performing a, at a positive reply X is performed and at a negative reply Y is performed. The program notation combining a LPNA based program notation A and these conditional instructions is denoted with A:C. A projection for the conditional construct instructions is found using second level instructions. Due to the use of second level instructions the projection can be given by replacing instructions without the need to update jump counters elsewhere in the program. The projection to second level instructions will be such that e.g.

$$\phi(b; +a\{; c; -d; \#\#0; \}\{; +e; \#\#4; \}; f) = b; -a\#\#7; c; -d; \#\#0; \#\#10; +e; \#\#4; \#\#10; f.$$

The general pattern of this projection is as follows: the conditional header instructions are mapped onto a sequence of two instructions, one performing the test and the second instruction containing the second level jump to the position following the projected separator instruction. The separator instruction is mapped to a jump to the position following the projected end of construct instruction, and the end of construct instruction is mapped to a skip, i.e., to a jump to the position thereafter. For this projection to work out some parsing of the program is needed in order to find out which separator instruction matches with which conditional construct header instruction and which end of construct instruction.

The annotated brace instructions needed for the projection of the conditional construct instructions are as follows: $+a\{k,\ -a\{k,\ k\}\{l,\ k\}$ with k, l natural numbers. These numbers indicate the position of instructions that c ontain the corresponding opening or closing brace, or 0 if such an instruction cannot be found in the program. A:Ca is the extension of a program notation A with annotated versions of the special conditional statements. An annotated version of the program given above is

$$b; +a\{6; c; -d; \#\#0; 2\}\{9; +e; \#\#4; 6\}; f.$$

The projection from A:Ca to A:SL is now obvious.

Projection 3.2. Let LPNA $\subseteq$ A.
ϕ : A:Ca $\to$ A:SL is defined by $\phi(u_1; \ldots; u_n) = \phi_1(u_1); \ldots; \phi_n(u_n)$ where

$$\begin{array}{rcl} \phi_i(+a\{k) & = & -a\#\#k+1 \\ \phi_i(-a\{k) & = & +a\#\#k+1 \\ \phi_i(l\}\{k) & = & \#\#k+1 \\ \phi_i(k\}) & = & \#\#i+1 \end{array}$$

for $k > 0$ and,

$$\begin{array}{rcl} \phi_i(+a\{0) & = & -a\#\#0 \\ \phi_i(-a\{0) & = & +a\#\#0 \\ \phi_i(l\}\{0) & = & \#\#0 \\ \phi_i(0\}) & = & \#\#0 \end{array}$$

The other instructions remain unmodified.

The projection of A:C to A:Ca involves a global inspection of the entire A:C program. There is room for confu sion, for instance, in the case

$$P = a; \}; +b\{; c; \}\{; d; \}\{; e; \{.$$

In spite of the fact that the program P is not a plausible outcome of programming in LPNA:C, an annotated version of can be established as

$$a; 0\}; +b\{5; c; 3\}\{7; d; 5\}\{0; e; \{0,$$

which contains all semantic information needed for a projection.

Projection 3.3. Let LPNA $\subseteq$ A.
ϕ : A:C $\to$ A:Ca is defined by $\phi(u_1; \ldots; u_n) = \phi(u_1); \ldots; \phi(u_n)$ where

1. $\phi(+a\{) = +a\{k$ and $\phi(-a\{) = -a\{k$
   with k the instruction number of the corresponding separator instruction if it exists and 0 otherwise,

2. $\phi(\}\{) = k\}\{l$
   with k and l the instruction numbers of the corresponding header and end of construct instruction if they exist and 0 otherwise,
3. $\phi(\}) = k\}$
   with k the instruction number of the corresponding separator instruction if it exist and 0 otherwise.

The other instructions remain unmodified.

3.2. Linear projective syntax for while loops

The following three instructions support the incorporation of while loops in programs projectible to LPNA: $+a\{*, -a\{*$ (*while loop header* instructions) and $*\}$ (*while loop end* instruction). This gives the program notation A:W. As in the case of conditional constructs, while constructs will be projected using annotated versions.

A:Wa allows the annotated while loop instructions $+a\{*k$, $-a\{*k$ and $k*\}$. The projection of annotated while loop instructions is again obvious.

Projection 3.4. Let LPNA $\subseteq$ A.
ϕ : A:Wa $\to$ A:SL is defined by $\phi(u_1; \ldots; u_n) = \phi_1(u_1); \ldots; \phi_n(u_n)$ where

$$\begin{array}{rcl} \phi_i(+a\{*k) & = & -a\#\#k+1 \\ \phi_i(-a\{*k) & = & +a\#\#k+1 \\ \phi_i(k*\}) & = & \#\#k \end{array}$$

for $k > 0$ and,

$$\begin{array}{rcl} \phi_i(+a\{*0) & = & -a\#\#0 \\ \phi_i(-a\{*0) & = & +a\#\#0 \\ \phi_i(0*\}) & = & \#\#0 \end{array}$$

The other instructions remain unmodified.

The introduction of annotated while braces by means of an annotating projection follows the same lines as for the case of conditional statement instructions.

Projection 3.5. Let LPNA $\subseteq$ A.
ϕ : A:W $\to$ A:Wa is defined by $\phi(u_1; \ldots; u_n) = \phi(u_1); \ldots; \phi(u_n)$ where

1. $\phi(+a\{*) = +a\{*k$ and $\phi(-a\{*) = -a\{*k$
   with k the instruction number of the corresponding separator instruction if it exists and 0 otherwise, and
2. $\phi(*\}) = k*\}$
   with k the instruction number of the corresponding separator instruction if it exist and 0 otherwise.

The other instructions remain unmodified.

Notice that a projection of the extension A:C of an LPNA based program notation A obtained by a simultaneous introduction of conditional and while constructs can be given by a concatenation of the projections defined above by starting either with conditional or while instructions. Thus

A:CW = (A:C):W $\to$ (A:C):Wa $\to$ (A:C):SL $\to$ A:C $\to \cdots \to$ A.

We end this section with an example containing conditional and while construct instructions.

Example 3.2. Let $P = +a\{; b; +c\{*; d; *\}; \}\{; e; f; \}$. P can be read as do a and continue with e and f and terminate if a returns `false`; otherwise do b and repeat $c; d$ as long as the last reply for c was positive, and terminate after the first negative reply to an execution for c. We may view P as a program in (LPNA:C):W. The annotated version of P given by the above projections is $+a\{; b; +c\{*5; d; 3*\}; \}\{; e; f; \}$ and projection to (LPNA:C):SL yields

$+a\{; b; -c\#\#6; d; \#\#3; \}\{; e; f; \}$.

Projecting to LPNA:C we obtain

$+a\{; b; -c; \#\#7; d; \#\#3; \}\{; e; f; \}.$

We now proceed with the projection of the conditional constructs. The annotated version is

$+a\{7; b; -c; \#\#7; d; \#\#3; 1\}\{10; e; f; 7\}$

and projection to LPNA:SL yields

$-a\#\#8; b; -c; \#\#7; d; \#\#3; \#\#11; e; f; \#\#11.$

Finally, projecting to LPNA we obtain

$-a; \#\#9; b; -c; \#\#8; d; \#\#4; !; e; f; !.$

3.3. Linear projective syntax for labels and goto's

Labels have been introduced in the program algebra setting in [BeL02] in the program notation PGLDg. Here, however, labels will have the form $[s]$, with s a non-empty alphanumerical string, i.e., a sequence over the alphabet $\{a, \ldots, z, A, \ldots, Z, 0, \ldots, 9\}$; $\#\#[s]$ and $\#\#[s][t]$ are the corresponding single or chained goto instructions. The intended meaning is that a label is executed as a skip. A single goto stands for a jump to the leftmost instruction containing the corresponding label if that exists and a termination instruction otherwise, and a chained goto of the form $\#\#[s][t]$ stands for a jump to the leftmost instruction containing the label $[s]$ followed by a jump to the leftmost instruction thereafter containing the label $[t]$ assuming hat they both exist and termination otherwise. The label occurrence that serves as the destination of a single or chained goto is called the *target* occurrence of the goto instruction.

In order to provide a projection for A:GL (program notation A with goto's and labels) the introduction of annotated goto's is useful. $\#\#[s]m$ ($\#\#[s][t]m$) represents the goto instruction $\#\#[s]$ ($\#\#[s][t]$) in a program where the target label is at position m. $\#\#[s]0$ ($\#\#[s][t]0$) represents the case that no target label exists. The projection from A:GLa (A with annotated goto's) to A and from A:GL to A:GLa are again obvious.

Projection 3.6. Let LPNA $\subseteq$ A.

1. ϕ : A:GLa $\to$ A is defined by $\phi(u_1; \ldots; u_n) = \phi_1(u_1); \ldots; \phi_n(u_n)$ where
   (a) $\phi_i([s]) = \#\#i + 1$,
   (b) $\phi_i(\#\#[s]m) = \phi_i(\#\#[s][t]m) = \#\#m$.

   The other instructions remain unmodified.
2. ϕ : A:GL $\to$ A:GLa is defined by $\phi(u_1; \ldots; u_n) = \phi(u_1); \ldots; \phi(u_n)$ where $\phi(\#\#[s]) = \#\#[s]m$ and $\phi(\#\#[s][t]) = \#\#[s][t]m$ with m the instruction number of the target label if it exists and 0 otherwise. The other instructions remain unmodified.

The projections discussed in this section are summarized in Figure 1.

4. Extensions using state machines

In some cases a program needs to make use of a data structure for its computation. This data structure is active during the computation only and helps the control mechanisms of the program. Such a data structure is called a *state machine* [BeP02]. A formalization of this matter involves a refinement of the syntax for basic actions. Any state machine action used in a program is now supposed to consist of two parts, respectively the *focus* and the *method*, glued together with a period that is not permitted to occur in either one of them. For the state machine only the method matters because the focus is used to identify to which state machine (or other system component) an atomic instruction is directed. We denote the extension of a program in notation A by the instructions of state machine SM by A/SM. A state machine action a used in the extension will be written as $sm.a$.

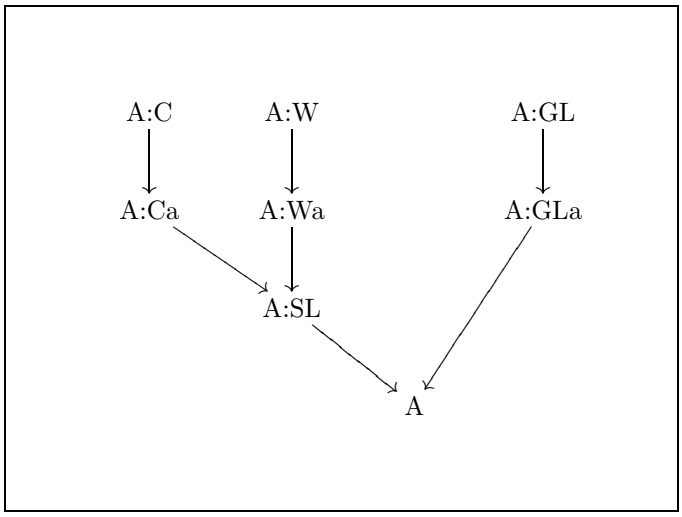

Fig. 1. Basic extensions of an LPNA based program notation A

4.1. Instruction counter stacks

ICS has an inf inite set of states, one for each stack of natural numbers. The basic actions performed by ICS are:

- *pop*. The pop action fails at an empty stack producing a negative reply; otherwise it succeeds changing the state of the stack and producing a positive reply.
- $push(n)$. The push action always succeeds changing the stack accordingly and producing a positive reply
- $topeq(n)$[2]. This action fails at an empty stack or if the top of the stack is unequal to n; otherwise it returns `true`.

A typical example of a program written in LPNA/ICS:C is

$$P = +a\{;ics.push(0);\}\{;-ics.pop;!;\};\#\#1.$$

When running, P can remember the number of positive replies it has obtained on its actions a, enabling it to produce termination after exactly as many subsequent negative replies. This implies that P has infinite state behavior. As a consequence, it is impossible to project LPNA/ICS:C to any notation making use of finite state state machines only. The key application of ICS is to obtain projections for program constructs involving recursion. In preparation of that application dynamic jumps for ICS will be needed.

With an instruction counter stack the *dynamic jump* $\#\#ics.pop$—where the counter is based on the top of the current stack—becomes available. The program notation that combines an extension A of LPNA with dynamic jumps is denoted with A:DJ. The projection to A/ICS:C:SL takes into account that in a program $u_1;\ldots;\#\#ics.pop;\ldots;u_k$ at most $k+1$ different values of the jump counter are needed.

Projection 4.1. Let LPNA $\subseteq$ A.
ϕ : A:DJ $\to$ A/ICS:C:SL is defined by $\phi(u_1;\ldots;u_k) = \phi(u_1);\ldots;\phi(u_k)$ where $\phi(\#\#ics.pop) = \psi_k$ and

$$\begin{aligned} \psi_1 &= +ics.topeq(1)\{;\sim ics.pop\#\#1;\sim\}\{;\sim!;\sim\} \\ \psi_{n+1} &= +ics.topeq(n+1)\{;\sim ics.pop\#\#(n+1);\sim\}\{;\sim \psi_n;\sim\} \end{aligned}$$

Here $\sim\psi$ denotes the instruction rij obtained from ψ by turning the first instruction of ψ into a second level instruction. All other instructions remain unmodified.

Having dynamic jumps available, we can now introduce recursion. The instruction pair $R\#\#k$, $\#\#R$ represents recursion in the following way: if a *returning jump instruction* $R\#\#k$ occurring at position i is executed

[2] Since basic actions are supposed to return boolean values, we have replaced the usual *top* operation—returning the top of the stack if it exists— by an indicator function.

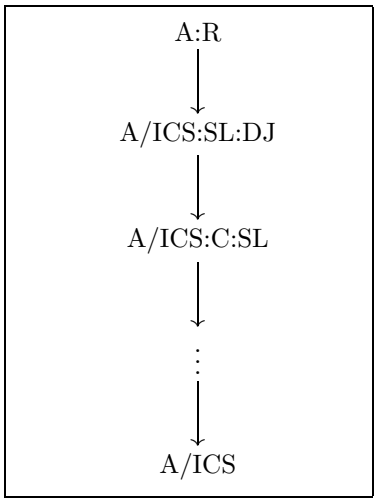

Fig. 2. Extensions using instruction counter stacks

a jump to instruction k is made, moreover the instruction counter $i+1$ is placed on an instruction counter stack known under focus *ics*. Whenever a *return instruction* $\#\#R$ is executed a jump is performed to the top of the stack which is simultaneously popped .

A:R is the program notation A augmented with returning jump instructions of the form $R\#\#k$ and return instructions $\#\#R$. A projection to A/ICS:SL:DJ can be given in the following way.

Projection 4.2. Let LPNA $\subseteq$ A.
ϕ : A:R $\rightarrow$ A/ICS:SL:DJ is defined by $\phi(u_1;\ldots;u_k) = \phi_1(u_1);\ldots;\phi_k(u_k)$ where

$$\begin{array}{rcl} \phi_i(R\#\#k) & = & ics.push(i+1)\#\#k \\ \phi_i(\#\#R) & = & \#\#ics.pop \end{array}$$

All other instructions remain unmodified.

The projections discussed in this subsection are summarized in Figure 2. Notice that (A/ICS:SL)/ICS:C:SL= A/ICS:C:SL. The final projection to A/ICS can be obtained by chaining the appropriate projections discussed in the preceding section.

4.2. Evaluation stacks

Besides the use of a stack to store instruction counters needed for the projection of recursive jump and return instructions, another very prominent use of a stack is that of an *evaluation stack* which contains intermediate results of the computation of some value in a data type.

For simplicity, we restrict ourselves here to a single primitive type consisting of all alphanumerical strings of bounded length and accompanied by a family of operators. Each operator f has an arity, which is a natural number that determines its number of arguments. In addition to the elements of primitive type, there are references uniquely denoting class instances. Primitive elements and references will be called *values*.

The name of the state machine as outlined below is EVS. It maintains a stack, a heap[3] and a value pool. The stack stores primitive types, references to objects and instruction counters; instance data and the class of objects are memorized in the heap. The value pool consists of finitely many pairs with the first element being the *spot* relative to the value pool and the second element its content. There are two distinguished spots: *this* and *that*. *this* contains the instances for object oriented instance method calls and *that* the results of subcomputations. The basic actions performed by EVS are:

- *compute*(f,k) (with f the name of an operator with arity k). It is assumed that the arguments of the

[3] Here we treat the heap as a rather abstract notion. For a concrete implementation the reader is referred to the appendix.

function are stored on the stack. If there are too few values on the stack the action fails and the state is unchanged. If there are sufficiently many values the reply is positive and the value $f(v_1, ..., v_k)$ is placed on the stack after removing all k arguments from it.

- *create*. This basic action allocates a new reference and puts it on top of the stack. If a new reference is not available this action fails.
- $store(s)$. The precondition of this basic action is that the stack is non-empty; otherwise it fails. If the stack is non-empty its top is removed and made to be the content of spot s. The previous content of s, if any, is lost in the process.
- $load(s)$. This basic action takes the content of the spot s, which must have been introduced at an earlier stage, and pushes that on the stack. A spot is defined if it has been assigned a value by means of a *store* instruction at some stage. If the spot is still undefined the action fails. This is a copy instruction because the value is still the content of s as well after successful execution.
- $getfield(F)$. The precondition of this action is that the top of the stack is a reference denoting an instance with field F; otherwise it fails. If the precondition is met, the value of field F of the instance denoted by the top is pushed from the heap on the stack after having removed the top.
- $setfield(F)$. The precondition of this action is that the top of the stack is a reference followed by a value; otherwise it fails. If the precondition is met, the reference and the value are popped from the stack and field F of the instance denoted by the popped reference is set to the popped value.
- $instanceof(C)$. The precondition of this action is that the top of the stack is a reference; otherwise it fails. If the precondition is met, this action returns `true` if the class of the instance denoted by the reference is C, otherwise it returns `false`. Afterwards the reference is taken from the stack.
- $setclass(C)$. Again, the precondition is that the top of the stack is a reference; otherwise the action fails. If the precondition is met, the class of the instance denoted by the reference is set to C. Afterwards the reference is popped.
- *pop*. This action fails if the stack is empty. Otherwise its succeeds and removes the top from the stack while giving a positive reply.

The combination of an evaluation stack and recursion is easily projected. The disadvantage of these projections lies in the fact that this form of projection introduces an instruction counter stack independently of the evaluation stack. This independence of the two stacks immediately gives rise to a Turing complete program notation in spite of the finiteness of the value pool. For that reason a different design will be developed that makes use of a single stack only at the cost of expressive power but—as will be demonstrated in the next section—sufficiently expressive to provide a projection for method calls in an object-oriented setting.

4.3. Stack fusion

ICS and EVS can be integrated in such a way that both features can be used from essentially a single stack. There are several reasons to pursue this line of design:

- Single stack execution environments have less computing power and admit a more thorough automatic analysis for that reason than double stack environments.
- It is important to understand which features in program notations require a more expressive execution architecture. For instance security features may involve a stackwalk, and multi-threading may involve multiple stacks, both requiring a Turing complete model of computation.
- There is a variety of possibilities for integrated stack design and stack use which provides insight in different forms of method calls and parameter mechanisms. These designs capture fundamental intuitions of program notation design and of program execution.

The fused stack S stores instruction counters and values. Its basic actions are:

- $compute(f, k)$ (with f the name of an operator with arity k). It is assumed that k values are stored consecutively on the stack. If there are too few values on top of the stack the action fails and the state is unchanged. If there are sufficiently many values the reply is positive and the value $f(v_1, ..., v_k)$ is placed on the stack after removing all k arguments from it.

- *create*. This basic action allocates a new reference and puts it on top of the stack. If a new reference is not available this action fails.
- *store*(s). The precondition of this basic action is that the stack is non-empty; otherwise it fails. If the stack is non-empty its top is removed and made to be the content of spot s. The previous content of s, if any, is lost in the process.
- *load*(s). This basic action takes the content of the spot s, which must have been introduced at an earlier stage, and pushes that on the stack. A spot is defined if it has been assigned a value by means of a *store* instruction at some stage. If the spot is still undefined the action fails. This is a copy instruction because the value is still the content of s as well after successful execution.
- $getfield(F)$. The precondition of this action is that the top of the stack is a reference denoting an instance with field F; otherwise it fails. If the precondition is met, the value of field F of the instance denoted by the top is pushed from the heap on the stack after having removed the top.
- $setfield(F)$. The precondition of this action is that the top of the stack is a reference followed by a value; otherwise it fails. If the precondition is met, the reference and the value are popped from the stack and field F of the instance denoted by the popped reference is set to the popped value.
- $instanceof(C)$. The precondition of this action is that the top of the stack is a reference; otherwise it fails. If the precondition is met, this action returns `true` if the class of the instance denoted by the reference is C, otherwise it returns `false`. Afterwards the reference is taken from the stack.
- *setclass*(C). Again, the precondition is that the top of the stack is a reference; otherwise the action fails. If the precondition is met, the class of the instance denoted by the reference is set to C. Afterwards the reference is popped.
- *pop*. This instruction fails if the stack is empty. Otherwise it succeeds and removes the top from the stack while giving a positive reply.
- *push*(n). The push action always succeeds changing the stack accordingly and producing a positive reply.
- *topeq*(n). This action returns `true` if the top of the stack equals n; otherwise it returns `false`.
- *down*(n). This action places the top just below the n-th position from above after removing the top. Thus after *down*(n) the top has migrated to the $n+1$-th position from above. If that is impossible the action fails, a negative reply is given and no change is made to the stack.

The projection of recursion is now a simple modification of the projections via dynamic jumps given in Projection 4.1 and 4.2.

Projection 4.3. Let LPNA $\subseteq$ A.

- ϕ : A:DJ $\rightarrow$ A/S:C:SL is defined by $\phi(u_1;\ldots;u_k) = \phi(u_1);\ldots;\phi(u_k)$ where $\phi(\#\#s.pop) = \psi_k$ and

$$\begin{array}{rcl} \psi_1 & = & +s.topeq(1)\{;\sim s.pop\#\#1;\sim\}\{;\sim !;\sim\} \\ \psi_{n+1} & = & +s.topeq(n+1)\{;\sim s.pop\#\#(n+1);\sim\}\{;\sim \psi_n;\sim\} \end{array}$$

  All other instructions remain unmodified.

- ϕ : A:R $\rightarrow$ A/S:SL:DJ is defined by
  $\phi(u_1;\ldots;u_k) = \phi_1(u_1);\ldots;\phi_k(u_k)$ where

$$\begin{array}{rcl} \phi_i(R\#\#k) & = & s.push(i+1)\#\#k \\ \phi_i(\#\#R) & = & \#\#s.pop \end{array}$$

All other instructions remain unmodified.

In addition to the dynamic jump, we shall also consider a *dynamic chained goto* instruction $\#\#[instanceof][M]$ where the first goto depends on the class of the instance *this*. The projection from A:DCG (A with dynamic chained goto's) to A/S:C:GL:SL can be given in the following way.

Projection 4.4. Let LPNA $\subseteq$ A.

ϕ : A:DCG $\to$ A/S:C:GL:SL is defined by $\phi(u_1;\ldots;u_k) = \phi(u_1);\ldots;\phi(u_k)$ where

$$\begin{aligned}\phi(\#\#[instanceof][M]) \;=\;& s.load(this);\\ & \sim +instanceof(C1)\{;\\ & \sim s.pop;\sim \#\#[C1][M];\\ & \sim\}\{;\\ & \sim +instanceof(C2)\{;\\ & \sim s.pop;\sim \#\#[C2][M];\\ & \sim\}\{;\cdots\\ & \sim +instanceof(Cn-1)\{;\\ & \sim s.pop;\sim \#\#[Cn-1][M];\\ & \sim\}\{;\\ & \sim s.pop;\#\#[Cn][M];\sim\};\cdots;\sim\}\end{aligned}$$

Here $C1,\ldots,Ck$ are the classes introduced by *setclass* instructions in $u_1;\ldots;u_k$. All other instructions remain unmodified.

Recursion using returning jumps and return instructions can be made more expressive by means of parameters. In the sequel we write $Rn\#\#k$ $(n,k\in\mathbb{N})$ for returning jump instructions with n parameters. Given a program notation A the extension by parametrized returning jumps and returning return instructions is denoted A:Rp.

The part of the computation taking place between a returning jump and its corresponding return instruction may be called a subcomputation. Parameters are values that serve as inputs to a subcomputation. They are put in place before executing a returning jump, and the program part executed as a subcomputation 'knows' where to find these parameters. Various strategies can be imagined for arranging the transfer of parameters during returning jump and return instructions. Here we have chosen for an automatic parameter transfer to and from the stack to local spots.

The n arguments are placed in the value pool at spots $arg1$,..,$argn$ and the return value—if there is one—will be expected at the spot *that*. In some cases a special parameter may be placed in spot *this*. *this* is used for the special parameter that plays the role of the target instance for an object oriented instance method call. Returning jump instructions with instance and n parameters will be denoted $IRn\#\#k$.

In preparation of the transfer of the arguments and the instance parameter to their respective spots the contents of these spots have to be placed in the stack in order to allow recovery of their contents after returning from the returning jump. Just after the returning jump and just before the return instruction the top of the stack contains the instruction counter followed by the contents of *this* and the n arguments in decreasing order. The projection of the return instruction will now involve the recovery of the several spot values before the returning jump from the data that were placed on the stack.

Projection 4.5. Let LPNA $\subseteq$ A.
ϕ : A:Rp $\to$ A/S:SL:DJ is defined by $\phi(u_1;\ldots;u_k) = \phi_1(u_1);\ldots;\phi_k(u_k)$ where

$$\begin{aligned}\phi_i(Rn\#\#k) \;=\;& \sim s.load(arg1);\sim s.down(n);\\ & \vdots\\ & \sim s.load(argn);\sim s.down(n);\\ & \sim s.push(i);\sim s.down(n);\\ & \sim s.store(arg1);\cdots;\sim s.store(argn);\sim \#\#k;\\ & s.store(argn);\sim s.store(argn);\cdots;\sim s.store(arg1)\\ \\ \phi_i(IRn\#\#k) \;=\;& \sim s.load(arg1);\sim s.down(n+1);\\ & \vdots\\ & \sim s.load(argn);\sim s.down(n+1);\\ & \sim s.load(this);\sim s.down(n+1);\\ & \sim s.push(i);\sim s.down(n+1);\\ & \sim s.store(this);\sim s.store(arg1);\cdots;\sim s.store(argn);\\ & \sim \#\#k;s.store(this);\sim s.store(argn);\cdots;\sim s.store(arg1)\\ \phi_i(\#\#R) \;=\;& \#\#s.pop\end{aligned}$$

All other instructions remain unmodified.

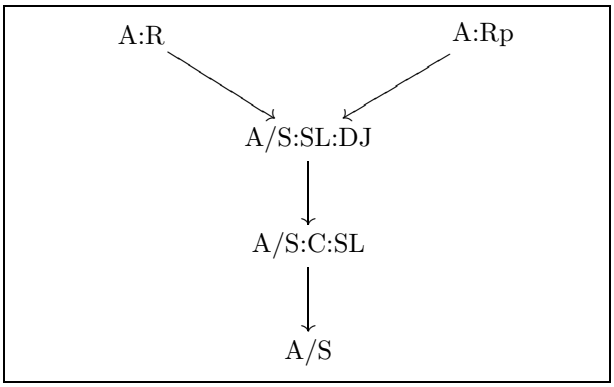

Fig. 3. Projections of recursion and parametrized recursion with automatic parameter and result transfer using a fused stack

It is important to notice that for the projection of a specific program only a finite number of $down(i)$ instructions is used. This implies that the stack manipulations do not reach the expressive power of a full stackwalk, but rather may be simulated (per program) from a stack and a finite memory state machine, or more easily obtained using a bounded value buffer.

The projections of recursion instructions are summarized in Figure 3. The final projection to A/S can be obtained by chaining the appropriate projections discussed in the preceding sections.

5. Intermediate level programs

The design of instructions can be viewed as proceeding in three phases. Until this point only so-called low level instructions have been designed. Low level instructions have either fixed projections or their projection trivially depends on the program context. Program context dependence occurs for instance with a closing brace instruction for which the projection may depend on the index of its corresponding opening brace instruction. The second design phase introduces an intermediate language for high-level programming constructs.

In this section we proceed with the second design phase. A typical ILN program consists of a sequence of labeled class parts. E.g., an ILN program with 4 classes looks as follows:

$$[C1]; CB1; \ldots; [C4]; CB4.$$

The subprograms $CB1, \ldots, CB4$ are so-called class bodies, which consist of zero or more labeled method body parts with end markers

$$[M1]; MB1; end; \ldots; [Mk]; MBk; end.$$

In addition to the usual labels, ILN has the following instruction set.

new(C) This instruction pushes a new reference to an instance of class C onto the stack.

stack push The instruction $E \Rightarrow$ represents pushing an entity with name E onto the stack. E may be a single non-empty alphanumerical string or of the form $E.F$, i.e., a string consisting of two alphanumerical parts glued together with a period.

top from stack The instruction $\Rightarrow E$ takes the top from the stack and places that entity on the place denoted with E. Again, E may be a single non-empty alphanumerical string or of the form $E.F$.

method end marker The method end marker *end* serves as the end of a method.

function call A function call $fc(f, n)$ represents the call of the function f from a given operator family with arity n. The algorithmic content of an operator call is not given by the program that contains it.

class method call A class method call instruction has the form $mc(C, M, n)$. Here M is a method, C a class and n is the number of arguments. We tacitly assume that the names occurring in the stack push and pop instructions of M are amongst $arg1, \ldots, argn$.

instance method call An instance method call instruction has the form $mc(M, n)$. Here M is a method

and n is the number of arguments. We tacitly assume that the names occurring in the stack push and pop instructions of M are amongst $this, arg1, \ldots, argn$.

ILN instructions can be projected to LPNA/S:DCG:GL:Rp. This projection carries with it an implicit runtime model. All intermediate language notations deal with a stack machine model. The degrees of freedom concern the precise details of argument transfer before and after recursion and argument protection during a recursive call. Many alternatives can be imagined for that model. Here we use the state machine S for an evaluation stack integrated with an instruction counter stack in fused style and accessible under focus 's'.

Projection 5.1. $\phi : \mathrm{ILN} \to \mathrm{LPNA/S:DCG:GL:Rp}$ is defined by $\phi(u_1; \ldots; u_k) = \phi_1(u_1); \ldots; \phi_k(u_k)$ where

$$\begin{array}{rcl}
\phi_i(new(C)) & = & s.create; s.setclass(C) \\
\phi_i(E \Rightarrow) & = & s.load(E) \\
\phi_i(\Rightarrow E) & = & s.store(E) \\
\phi_i(E.F \Rightarrow) & = & s.load(E); s.getfield(F) \\
\phi_i(\Rightarrow E.F) & = & s.load(E); s.setfield(F) \\
\phi_i(end) & = & \#\#R \\
\phi_i(fc(f,n)) & = & s.compute(f,n) \\
\phi_i(mc(C,M,n)) & = & Rn\#\#2k+l+i+2; \#\#2k+l+i+3; \#\#[C][M] \\
\phi_i(mc(M,n)) & = & IRn\#\#2k+l+i+2; \#\#2k+l+i+3; \\
 & & \#\#[instanceof][M]
\end{array}$$

Here k is the number of method calls and l the number of creation and field pushing and popping instructions occurring in $u_1; \ldots; u_{i-1}$. In a setting with relative jumps one can replace the complex absolute jumps by relative jumps of length 2 and 3, respectively. Labels remain unmodified.

Example 5.1. A simple ILN program P and its projection are given below. P consists of a single class part labeled $[C]$ with method parts $[CC]$, $[get]$ and $[M]$. $[CC]$ is a class constructor initializing the F field of any object of this class with default value 0, $[get]$ is an instance method returning the value of the F field of the calling object, and $[M]$ is a static method creating two objects of this class—X and Y—by calling the constructor for X but setting the F field of Y by calling $[get]$ with parameter X.

$[C];$
$\quad [CC]; fc(0,0); \Rightarrow this.F; end;$
$\quad [get]; arg1.F \Rightarrow; \Rightarrow that; end;$
$\quad [M]; new\ C; \Rightarrow X; X \Rightarrow; mc(CC,0);$
$\quad\quad new\ C; \Rightarrow Y;$
$\quad\quad X \Rightarrow; mc(C,get,1); that \Rightarrow; \Rightarrow Y.F; end$

Projection into LPNA/S:DCG:GL:RP yields

$[C];$
$\quad [CC]; s.compute(0,0); s.load(this); s.setfield(F); \#\#R;$
$\quad [get]; s.load(arg1); s.getfield(F); s.store(that); \#\#R;$
$\quad [M]; s.create; s.setclass(C); s.store(X); s.load(X); IR0\#\#19; \#\#20;$
$\quad\quad \#\#[instanceof][CC];$
$\quad\quad s.create; s.setclass(C); s.store(Y);$
$\quad\quad s.load(X); R1\#\#26; \#\#27; \#\#[C][get]; s.load(that); s.load(Y);$
$\quad\quad s.setfield(F); \#\#R$

6. A high-level program notation HLN

In the third phase we deviate from the linear projective syntax paradigm and introduce instructions at a level close to the high-level program notations used by human programmers and computer software users.

In this section a toy high-level program notation HLN is developed. Its meaning is given by a compiler projection into ILN. Like ILN programs HLN programs are supposed to have a certain fixed structure. In this structure a program is a series of class parts as depicted in Figure 4. Class parts consist of a number of segments. These segments can be of two kinds: instance field parts and method declarations. The compiler

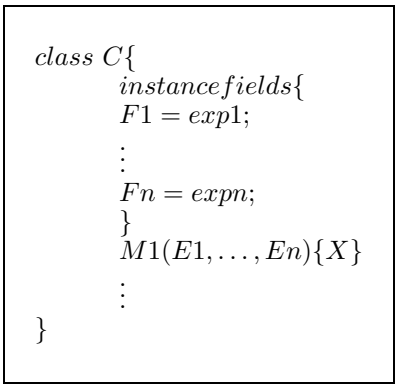

Fig. 4. The structure of class parts in HLN

projection will transform the list of instance field parts in a class to a class constructor for that class with label $[CC]$. Method declarations—which have the form $M(E1,\ldots,En)\{X\}$ and are named different from CC—will be projected to class method parts.

HLN has the following expressions.

$$exp := E \mid E.F \mid f(exp1,\ldots,expn) \mid I.M(exp1,\ldots,expn) \mid C.M(exp1,\ldots,expn)$$

with f an n-ary operator from a given operator family, I a class instance calling an n-ary instance method and $C.M(exp1,\ldots,expn)$ an n-ary class method call. Assignments and statements are of the form

$$\begin{array}{rcl} E & = & exp \\ E.F & = & exp \\ E & = & new\ C \\ E.F & = & new\ C \\ & return\ exp & \\ I.M(exp1,\ldots,expn) & & \\ C.M(exp1,\ldots,expn) & & \end{array}$$

with $exp, exp1,\ldots,expn$ expressions. The projections to ILN can be given by

$$\begin{array}{rcl} \phi(E = exp) & = & \phi(exp); \Rightarrow E \\ \phi(E.F = exp) & = & \phi(exp); \Rightarrow E.F \\ \phi(E = new\ C) & = & new\ C; mc(CC,0); \Rightarrow E \\ \phi(E.F = new\ C) & = & new\ C; mc(CC,0); \Rightarrow E.F \\ \phi(return\ exp) & = & \phi(exp); \Rightarrow that \\ \phi(I.M(exp1,\ldots,expn)) & = & \phi(exp1);\ldots;\phi(expn); I \Rightarrow; mc(M,n) \\ \phi(C.M(exp1,\ldots,expn)) & = & \phi(exp1);\ldots;\phi(expn); mc(C,M,n) \end{array}$$

where the expressions $exp, exp1,\ldots,expn$ are projected to values residing on top of the stack by

$$\begin{array}{rcl} \phi(E) & = & E \Rightarrow \\ \phi(E.F) & = & E.F \Rightarrow \\ \phi(f(exp1,\ldots,expn)) & = & \phi(exp1);\ldots;\phi(expn); fc(f,n) \\ \phi(I.M(exp1,\ldots,expn)) & = & \phi(exp1);\ldots;\phi(expn); I \Rightarrow; mc(M,n), that \Rightarrow \\ \phi(C.M(exp1,\ldots,expn)) & = & \phi(exp1);\ldots;\phi(expn); mc(C,M,n), that \Rightarrow \end{array}$$

The meaning of class parts can finally be given by

$$\begin{array}{rcl} \phi(class\ C\{X\}) & = & [C]; \phi(X) \\ \phi(instancefields\{F1 = exp1;\ldots;Fn = expn\}) & = & [CC]; \\ & & \phi(this.F1 = exp1); \\ & & \ldots \\ & & \phi(this.Fn = expn); \\ & & end \end{array}$$

$$\begin{array}{lcl}\phi(M(E1,\ldots,En)\{X\}) & = & [M];\\ & & E1\Rightarrow;\ldots;En\Rightarrow;\\ & & \phi(E1=arg1);\\ & & \ldots\\ & & \phi(En=argn);\\ & & \phi(X);\\ & & \Rightarrow En;\ldots;\Rightarrow E1;\\ & & end\end{array}$$

Note that the projection of HLN to ILN is dynamically correct. That is, the elements on the stack meet the requirements of the current instructions: whenever an instruction counter is expected, the top of the stack will contain one, and whenever n values are expected, the n uppermost elements will be values.

We end this section with a typical HLN program and its projection into ILN.

Example 6.1. Let P be the the following program

$$\begin{array}{l}class\ A\{\\ \quad instancefields\{F=0;\}\\ \quad get()\{return\ this.F;\}\\ \quad set(E)\{this.F=E;\}\\ \quad inc(E)\{this.set(this.get()+E);\}\\ \quad\}\end{array}$$

$$\begin{array}{l}class\ B\{\\ \quad get(E)\{return\ E.F;\}\\ \quad test()\{\\ \qquad X=new\ A;\\ \qquad Y=new\ A;\\ \qquad X.set(1);\\ \qquad X.inc(1);\\ \qquad Y.F=B.get(X);\\ \quad\}\end{array}$$

Assuming that $0, 1$ and $+$ are predefined operators, P can be projected into ILN by

$$\begin{array}{l}[A];\\ \quad [CC];fc(0,0);\Rightarrow this.F;end;\\ \quad [get];this.F\Rightarrow;\Rightarrow that;end;\\ \quad [set];E\Rightarrow;arg1\Rightarrow;\Rightarrow E;E\Rightarrow;\Rightarrow this.F;\Rightarrow E;end;\\ \quad [inc];E\Rightarrow;arg1\Rightarrow;\Rightarrow E;this\Rightarrow;mc(get,0);that\Rightarrow;E\Rightarrow;fc(+,2);\\ \qquad\quad this\Rightarrow;mc(set,1);end;\\ [B];\\ \quad [get];E\Rightarrow;arg1\Rightarrow;\Rightarrow E;E.F\Rightarrow;\Rightarrow that;\Rightarrow E;end;\\ \quad [test];\\ \qquad new\ A;mc(CC,0);\Rightarrow X;\\ \qquad new\ A;mc(CC,0);\Rightarrow Y;\\ \qquad fc(1,0);X\Rightarrow;mc(set,1);\\ \qquad fc(1,0);X\Rightarrow;mc(inc,1);\\ \qquad X\Rightarrow;mc(B,get,1);that\Rightarrow;\Rightarrow Y.F;\\ \qquad end\end{array}$$

7. Concluding remarks

At this point a simple program notation for object oriented programming has been provided with a linear projective syntax. Many more features exist and one might say that the project of linear projective syntax design has only been touched upon. Still the claim is made that the above considerations provide a basis for the design of projective syntax for much more involved program notations.

For the moment the primary application area of this work is considered to be the teaching of programming

concepts. With that objective in mind it is proposed as a reasonable claim that the given reconstruction of a subset of object oriented programming provides a systematic insight in the meaning of the constructions covered. The fact that these techniques are amenable for teaching, and more importantly that doing so provides a basis for the understanding of more involved aspects of programming cannot be taken for granted, however. Practical experience will have to clarify this claim.

References

[AFV01] L. Aceto, W.J. Fokkink, and C. Verhoef. Structural operational semantics. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, 197–292, North-Holland, 2001.

[BeB02] J.A. Bergstra and I. Bethke. Molecular dynamics. *Journal of Logic and Algebraic Programming*, 51(2):193–214, 2002.

[BeB03] J.A. Bergstra and I. Bethke. Polarized Process Algebra and Program Equivalence. In Jos C.M.Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors, *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003*, Lecture Notes in Comput. Sci. 2719, 1 –21, Springer, 2003.

[BeL00] J.A. Bergstra and M.E. Loots. Program algebra for component code. *Formal Aspects of Computing*, 12(1):1–17, 2000.

[BeL02] J.A. Bergstra and M.E. Loots. Program algebra for sequential code. *Journal of Logic and Algebraic Programming*, 51(2):125–156, 2002.

[BeP02] J.A. Bergstra and A. Ponse. Combining programs and state machines. *Journal of Logic and Algebraic Programming*, 51(2):175–192, 2002.

[Dij75] E.W. Dijkstra. Guarded commands, non-determinacy, and formal derivations of programs. *Commun. ACM*, 18:453–457, 1975.

[FeF87] M. Felleisen and D.P. Friedman. Control operators, the SECD machine, and the λ-calculus. In *Formal Description of Programming Concepts III, Proc. IFIP TC2 Working Conference*, Gl. Avernaes, 1986, 193–217, North-Holland, 1987.

[Gur95] Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.

[Har84] D. Harel. Dynamic logic. In *Handbook of Philosophical Logic*, volume II. D. Reidel Publishing Company, 1984.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, 1969.

[Kah87] G. Kahn. Natural semantics. In *STACS'87, Proc. Symp. on Theoretical Aspects of Computer Science*, Lecture Notes in Comput. Sci. 247, 22–39, Springer, 1987.

[Mog91] E. Moggi. Notions of computation and monads. *Information and Control*, 93:55–92, 1991.

[Mos90] P.D. Mosses. Denotational semantics. In *Handbook of Theoretical Computer Science*, volume B, chapter 11. Elsevier Science Publishers, Amsterdam; and MIT Press, 1990.

[Sev04] I.A. Severoni. Operational semantics of the Java Card Virtual Machine. *Journal of Logic and Algebraic Programming*, 58:3–25, 2004.

[SSB01] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine. Definition, Verification, Validation*. Springer, 2001.

8. Appendix: A Concrete heap

In this appendix we provide a more detailed and concrete, and therefore less abstract version of the heap. With BinSeq we denote the finite sequences of 0's and 1's. Three subsets of BinSeq are at the basis of the concrete heap:

- Loc is a finite or infinite set of so-called locations, (in practical cases it must be finite, in some formal modeling cases an infinite set of locations may be more amenable),
- Ic, the finite or infinite collection of instruction counters. Here the most plausible choice is that Ic contains binary forms of all natural numbers,
- Pval, the collection of primitive values, this is a finite set in most practical cases, just like Loc.

The concrete heap is used in combination with the stack. We will describe the setting in the fused case. On the stack one finds

- references, taking the form rw with w in Loc, e.g. r00110; the set of rw's with w in Loc is denoted Ref below,
- instruction counters taking the form iw with w in Ic, and
- primitive values taking the form w with w in Pval.

In any state of computation the spots have either no meaning (the initial state for all spots) or, if a spot has a meaning, that is either a reference rw or a primitive value.

Several collections of names are used: ClassNames, FieldNames and SpotNames. Any convention will do and these sets may overlap without generating confusion.

A state of the concrete heap consists of the following sets:

- Obj, a subset of Loc. Obj contains those locations that contain an object. Initially a computation will start with Obj empty.
- Classification, a subset of Obj $\times$ Classes, which is in fact a partial function form Obj to Classes.
- Fields, a subset of Obj $\times$ FieldNames $\times$ (Ref $\cup$ Pval). Again we assume that Fields is a partial function from Obj $\times$ Fieldnames to Ref $\cup$ Pval.
- Spots, a subset of SpotNames $\times$ (Ref $\cup$ Pval).

It is important to make a distinction between vital objects and garbage objects. Vital objects are those that can be reached from spots containing a reference and references positioned anywhere on the stack via zero or more selections of fields containing a reference. Non-vital objects are called garbage objects. After each method execution garbage objects are removed and so are their outgoing fields and classifications.

Now the operations of the heap can be interpreted as follows:

- *create*. If Obj = Loc then return `false`; otherwise find w in Loc - Obj, place rw on the top of the stack, put w in Obj, and then return `true`.
- $getfield(f)$. If the top of the stack is not a reference or if the top of the stack contains rw, say, but Fields(rw, f) is not defined return `false`; otherwise remove the top of the stack, place Fields(rw,f) on the stack and return `true`.
- $setfield(f)$. If the top of the stack is a reference rw, say, and if it lies on top of a value v in Ref $\cup$ Pval, pop the two uppermost items from the stack, define Fields(rw, f) = v and return `true`; otherwise return `false`.
- $instanceof(C)$. If the top of the stack is not a reference return `false`; if the top of the stack contains rw, say, pop the stack, return `true` if Classification (rw)=C, and return `false` otherwise.
- $setclass(C)$. If the top of the stack is not a reference return `false`; if the top of the stack contains rw, say, define Classification(rw)=C, pop the stack and return `true`.

In this concrete heap a reference takes the form rw and spots may also be considered references. In other models of a heap references may work differently so this description of a concrete heap cannot be taken as an analysis of what constitutes a reference in general.