ELSEVIER

The Journal of Logic and Algebraic Programming 51 (2002) 125–156

THE JOURNAL OF LOGIC AND ALGEBRAIC PROGRAMMING

www.elsevier.com/locate/jlap

Program algebra for sequential code

J.A. Bergstra[a,b,*], M.E. Loots[a]

[a] *Programming Research Group, Faculty of Science, University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands*

[b] *Applied Logic Group, Department of Philosophy, Utrecht University, Heidelberglaan 8, 3584 CS Utrecht, The Netherlands*

Abstract

The jump instruction is considered essential for an adequate theoretical understanding of imperative sequential programming. Using basic instructions and tests as a basis we outline an algebra of programs, denoted PGA, which captures the crux of sequential programming. We single out a behavior extraction operator which assigns to each program a behavior. The meaning of the expressions of PGA is explained in terms of the extracted behavior. Using PGA a small hierarchy of program notations is developed. Projection semantics is proposed as a tool for the description of program semantics.

Keywords: Program algebra; Projection semantics; Behavior extraction

1. Introduction

The objective of this paper is to describe in a rather 'non-formal' and theoretical style the most simple and basic concept of a programming language that we have been able to discover.[1] We have primarily focussed on sequential programming, because it still seems to be the basis of programming, both in practice and in teaching. An algebra of sequential programs (named PGA) is used as the carrier for our development of ideas. The syntax of PGA serves as a very simple program notation. Other program notations are developed on the basis of this simple one.

For some text to be called a program, its being intelligible as an implementation of some objective or specification is not a criterion. As a consequence we have refrained from the use of any example programs that have intuitive content. For PGA we will provide one

* Corresponding author.

E-mail addresses: janb@science.uva.nl, jan.bergstra@phil.uu.nl (J.A. Bergstra), mloots@science.uva.nl (M.E. Loots).

[1] This paper is based on [2], reproducing most of its content. We refer to that text for additional motivation and further references. Several important technical aspects have been modified significantly, however. Notably postconditional composition has been introduced as a semantic tool and the notation #, in [2] used for the abort primitive instruction, can now be seen as an abbreviation for #0 (i.e., a jump of length zero).

1567-8326/02/$ - see front matter
PII: S1567-8326(02)00018-8

semantic model which maps programs into behaviors: BPPA (for basic polarized process algebra). PGA is so simple that it can be easily memorized together with its semantic equations.

Once PGA has been established, more complex programming languages can be introduced, based on PGA. Defining PGA first as a firm semantic basis for later languages means that their introduction can be achieved against reduced overhead costs. We will return to that matter in Section 7.1.

1.1. Where to classify program algebra?

A classification of program algebra as a subject in the overall domain of science seems to profit from a less conventional terminology. We propose the following classification for informatics (and related areas), suggesting that program science should be the container for program algebra:

Computer science. This subject deals with computers all the way down from the basic principles and limitations in connection with physics, chemistry and biology to the principles of mechanical engineering, circuit design and so on.

Program science. Covers programs and programming. The question 'what is a program?' belongs here, just as well as a wide range of engineering issues. Being programs, compilers and operating systems are covered by program science as well.

Information science. Information science regards computers and programs merely as tools, the use of these tools being the main topic. Crucial is the functionality of these ingredients in a larger context and in connection with goals and objectives outside computing. Requirements capture and all forms of behavioral description and assessment belong here.

Cognitive science. Cognitive science addresses the biological information processing systems as well as the artificial ones from a broad perspective, including all philosophical issues raised in connection with the human mind. Artificial intelligence is a part of cognitive science.

An important interface between the fields just mentioned is that between program science and information science. The key concept shared by these two fields is that of a behavior, not that of a program. In no way is it obvious that program science constitutes a part of computer science. Further the well-known phrases 'software engineering,' 'software technology,' 'computer programming' and 'programming methodology' each have a built-in bias towards construction (over contemplation) making them useless as a field descriptor in the long run.

Program science can be compared to logic. It 'owns' the concept of a program in the same way as logic 'owns' the concepts of formalized reasoning and proof.

1.2. Principles and terminology

An informal terminology is needed as a basis for our analysis. A program is 'in essence' a non-empty sequence of instructions. Finite or infinite sequences of so-called primitive instructions constitute the mathematical objects appropriate for representing programs. The domain of the program algebra PGA consists of a infinite set of instruction sequences. Primitive instructions have been designed to enable single pass execution of instruction sequences. Each instruction can be dropped after having been processed.

Single pass (primitive) instruction sequences are also called program objects. For the development of program algebra it would be useful to simply identify the concept of a program with single pass instruction sequences. Unfortunately there would then be too large a gap between the common use of the word 'program' and the one introduced here. Usually programs are considered texts or expressions. For that reason we will say that a program is a text (or expression, or even a picture) representing an instruction sequence. For a text to qualify as a program it must be known how to obtain the instruction sequence (program object) that it stands for. We will use the phrase 'program text' (or 'program expression') for programs in this sense, thereby emphasizing that a program is more than a mere text.

A method for generating a (single pass primitive) instruction sequence from a text is called a projection function. A program language (program notation) is a collection of texts together with a projection function transforming each of its elements to a particular instruction sequence.

The program algebra PGA provides one with a fairly minimal syntax for program expressions. An important parameter enters the picture: Σ, the collection of so-called basic instructions. With the help of Σ the collection PI_Σ of primitive instructions is defined. SPI_Σ denotes the sequences (finite as well as infinite) of instructions in PI_Σ. SPI_Σ serves as our collection of program objects. PGA_Σ is an algebra of programs representable by sequences of instructions from PI_Σ.

Program algebra induces an equivalence on program expressions over the syntax of PGA_Σ (PGA_Σ-expressions), two program expressions being equivalent if their values in SPI_Σ coincide. This equivalence is called 'instruction sequence equivalence'. For technical reasons the phrase 'instruction sequence congruence' will be used instead, however.

Structural equivalence (structural congruence) is a further identification on program expressions (and on instruction sequences). It is characterized below by means of a number of equation schemes.

Behavioral equivalence identifies two instruction sequences if they represent the same behavior upon execution. Unlike instruction sequence congruence and structural congruence, behavioral equivalence is not a congruence. It fails to be compatible with the operators of PGA. In other words: behavioral equivalence is not compositional. Behavioral equivalence is defined by means of a behavior extraction operator (denoted by $|-|$), transforming each instruction sequence into a behavior. Instruction sequences X and Y are behaviorally equivalent if $|X| = |Y|$. Instruction sequence equivalence implies structural equivalence and structural equivalence implies behavioral equivalence, but not the other way around.

1.3. So what is a program?

Program objects (elements of SPI_Σ) are programs in virtue of being single pass instruction sequences. PGA_Σ-expressions are programs on the basis of their natural interpretation in SPI_Σ. If a collection C of texts (or more general of graphical objects) admits a projection function (a function into SPI_Σ for an appropriate collection Σ of basic instructions), the elements of C may be considered programs as well. Most importantly a text can be a program only if it is known how the text represents a program object.

2. Program expression syntax

The syntax of program expressions in PGA is generated from five kinds of constants and two composition mechanisms. The constants are called primitive instructions. The primitive instructions are made from a parameter set Σ of so-called basic instructions.[2] These basic instructions may be viewed as requests to an environment to provide some service. It is assumed that upon every termination of the delivery of that service, some boolean value is returned that may be used for subsequent program control. The composition mechanisms of program algebra are the structuring 'features' of the programming language. The compositions are:

- concatenation of X and Y, written $X; Y$, and
- repetition of X, written X^ω.

The five forms of primitive instructions, are listed below, the negative test instructions having been included for the reason of symmetry.

Void basic instruction. All elements $a \in \Sigma$ are basic instructions; when executed these instructions may modify (have a side effect on) a state, a boolean value being generated in addition. The attribute void expresses that these instructions do not make use of the returned boolean value. After having performed an basic instruction a program has to enact its subsequent instruction. If that instruction fails to exist, inaction occurs. Inaction is a lack of activity without proper termination.[3]

It should be noticed that by definition each basic instruction is a primitive instruction (more specifically labeled as a void basic instruction) as well.

Termination instruction. The instruction ! indicates termination of the program. It will not lead to any further effects on the state, and it will not return any value.

Positive test instruction. For all instructions $a \in \Sigma$ there is the positive test instruction $+a$. If $+a$ is performed by a program, the state is affected according to a, and as a result of that process, a boolean value is produced and is returned to the program (or the processor running the program).

The sequence of remaining instructions is performed in case `true` was returned. If there are no remaining instructions execution cannot continue and inaction occurs. If `false` was returned after a was performed, the next instruction is skipped and execution proceeds with the instruction following the instruction that was skipped. In that case (i.e. when `false` was returned) at least two instructions must be present following $+a$; otherwise execution cannot continue and inaction occurs.

Negative test instruction. For all basic instructions $a \in \Sigma$ there is the negative test instruction $-a$. If $-a$ is performed by a program, the state is affected according to a, after which the remaining sequence of instructions is performed in case `false` was returned. Again, if there are no remaining instructions inaction takes place. If `true` was returned after a was performed, the next instruction is skipped and execution proceeds with the instruction following the instruction that was skipped. In that case (i.e. when `true` was returned) at least two instructions should be present following $-a$; otherwise inaction will occur.

---

[2] Basic instructions are considered indivisible from the perspective (abstraction level) of the program. At a lower level of abstraction the basic instructions may comprise entire program executions for other programs.

[3] In a setting of concurrent processes a deadlock (often denoted with δ), is a typical example of inaction. Later on we will write D for an inactive behavior, thus emphasizing this connection.

Forward jump instructions. For any natural number k there is an instruction $\#k$ which denotes a jump of length k. We call k the counter of the jump instruction. If $k = 0$, this jump is to the instruction itself (zero steps forward). In this case inaction will result. If the counter of the jump equals 1, the instruction is a skip (i.e. it skips itself). The subsequent instruction will be executed next. In case there are no further instructions program execution becomes inactive. If the counter (k) exceeds 1, the effect of the execution of an instruction $\#k$ is to skip itself and the next $k - 1$ instructions. If there are not that many instructions left in the remaining part of the program, program execution will become inactive.

The collection of primitive instructions is denoted with PI_Σ. Variables $u, v, w, u_i, \ldots$, range over primitive instructions. Examples of program expressions are:[4] $a; b; c$, $b; +a; \#5; \#2; c; c$, $(a; -b)^\omega; \#2; !$, and $(\#1)^\omega$. Capitals $X, Y, Z, U, \ldots$ (often used with subscripts and/or superscripts) will be used as variables for program objects. These variables will also be used to range over program expressions, as long as this is not a cause for confusion.[5]

There is an unavoidable element of arbitrariness in the selection of these primitives. However, the underlying concepts are by no means arbitrary. Concatenation is a very common primitive, jump instructions are standard in assembly languages, and some form of conditional construct appears in most program notations. The particular form of our primitives (except ';') is new to the best of our knowledge. The overriding concern has been to simultaneously simplify syntax and semantics while maintaining the expressive power of arbitrary finite control.

3. Single pass instruction sequences

The program expressions can be interpreted by simply identifying programs that give rise to identical sequences of instructions. An unfolded sequence of PGA instructions will be called a single pass instruction sequence. During a run of the instruction sequence each instruction is visited at most once and is dropped after having been performed. The single pass instruction sequence is 'the program'. Therefore our concept of a program is a mathematical one. We will also use the phrase program object for a single pass instruction sequence.[6] An object can be called a program if it is a single pass instruction sequence or if it can be translated into a single pass instruction sequence while preserving its essential meaning. The latter is a subjective matter. A graphical program can serve as a piece of art. Translating it into a single pass instruction sequence may deprive it from its artistic qualities. In such a case the graphical structure is not a program.

An important principle for program object equivalence is extensionality:[7] if program objects X and Y have equal length and equal nth instructions for all natural $n > 0$, then the two program objects must be equal. As a notation for the nth instruction[8] of program X we

[4] Brackets for ; are omitted because it will be assumed to be associative.

[5] A systematic distinction between variables and meta-variables has not been made in order to simplify the presentation. This additional precision can easily be introduced by taking $\mathscr{X}, \mathscr{Y}, \mathscr{Z}, \mathscr{U}, \ldots$, as meta-variables ranging over program expressions.

[6] The term 'object' (in 'program object') corresponds to the use of 'object' in mathematics, rather than its use in object-oriented programming.

[7] Extensional equivalence always refers to an equivalence on a whole which is being inferred from equivalences on all of its parts.

[8] Here n must be a positive natural number.

propose: $i_n(X)$.[9] In program algebra the principle of extensionality gives rise to instruction sequence equivalence.

3.1. Program object equations

We present three equations and an equation scheme (parametrized by a positive natural number n) that identify representations of the same single pass instruction sequences. These equations are called 'program object equations'.

$(X; Y); Z = X; (Y; Z)$	(PGA1)
$(X^n)^\omega = X^\omega$	(PGA2)
$X^\omega; Y = X^\omega$	(PGA3)
$(X; Y)^\omega = X; (Y; X)^\omega$	(PGA4)

Here $X^1 = X$ and $X^{n+1} = X; X^n$. We will refer to these equations as PGA1, ...,PGA4 respectively.[10] The unfolding of a repetition can be derived: $X^\omega = (X; X)^\omega = X; (X; X)^\omega = X; X^\omega$. The associativity of concatenation implies (as usual) that far fewer brackets have to be used. We will use associativity whenever confusion cannot emerge.

We will now start using the term 'action'. An action takes place if an instruction is performed. Programs contain instructions, the meaning of a program is made up from actions, however. From now on we will make no distinction between 'action' and 'basic instruction', assuming that the reader can disambiguate the language when needed. The main reason for not strictly adhering to the distinction lies in the intention to use the same notations for both.

It is obviously impossible to perform infinitely many actions in finite time.[11] It follows that right cancellation is obtained if a left-hand argument of a concatenation contains an infinite repetition. This is expressed by the third equation.

3.1.1. Instruction sequence congruence

If two program expressions (program texts) can be shown to be equal by means of PGA1-4, the program expressions are said to be instruction sequence equivalent. The unfolded instruction sequence determined by a program is also called a program object (an element of SPI_Σ). Instruction sequence equivalence is in fact a congruence relation. In the case of PGA an equivalence ($\equiv$) is a congruence if for all X, X', Y, Y' the following implications hold: if $X \equiv Y$, then $X^\omega \equiv Y^\omega$, and if moreover $X' \equiv Y'$, then $X; X' \equiv Y; Y'$. We will write $X =_{\mathrm{isc}} Y$ if X and Y are instruction sequence congruent, often omitting the subscript if no confusion arises.

The completeness of the four program object equations for instruction sequence congruence is discussed in Section 3.2.2. Instruction sequence congruence allows some useful program transformations, in particular the transformation into the first canonical form.

---

[9] Defining equations for this operator are: $i_1(u) = u$, $i_1(u; X) = u$, $i_{n+1}(u) = \#0$, and $i_{n+1}(u; X) = i_n(X)$.

[10] In the setting of process algebra, repetition is called 'perpetual loop', and studied in [3]. PGA1, ...,PGA4 can all be derived from the axioms given in that paper.

[11] So-called Zeno executions are excluded. This point requires some care. The exclusion of Zeno executions regards computations in which an action is preceded by infinitely many actions. Such computations are excluded in many theories of computation on philosophical grounds. In theories not based on a total ordering of time the concept of a Zeno execution should be phrased in terms of causal precedence rather than temporal precedence. PGA contains no jump instruction enabling it to 'jump over' an infinite sequence of instructions.

3.2. Canonical forms for PGA: the first canonical form

Let X be a closed PGA expression. Then X can be (re)written into one of the following forms:

- Y, not containing repetition,[12] or
- $Y; Z^\omega$, with Y and Z not containing a repetition.[13]

In any of the above forms, we will call closed terms canonical terms (of the first form). The subterm Z in the second case is called the repeating part of the (canonical) expression. The proof of the existence of the first canonical form uses induction on the structure. The key case is this: $(U; X^\omega)^\omega = (U; X^\omega); (U; X^\omega)^\omega = U; (X^\omega; (U; X^\omega)^\omega) = U; X^\omega$. Examples of identities between terms and terms in the first canonical form:

- $a^\omega; b; !; c =_{\mathrm{isc}} a; a^\omega$,
- $+b; (!; c; -b; (c; \#25; a; !)^\omega)^\omega =_{\mathrm{isc}} +b; !; c; -b; (c; \#25; a; !)^\omega$.

Confusion may arise around the notion of the length of a repeating part and that of a period. The repeating part is a portion of a syntactic expression. A program object, however, is periodical if it is the interpretation of a canonical term with infinite length. Its period is the shortest length of a repeating part of a canonical form representing the object.

3.2.1. Decidability of instruction sequence congruence

It should be noticed that program object equivalence ($=_{\mathrm{isc}}$) is a decidable matter for closed program expressions. Decidability can for instance be seen by writing two expressions in first canonical forms. If one of the two (first) canonical forms has finite length, the decision is made on obvious grounds because the two sequences must be identical. Otherwise both repeating parts can be made as short as possible with the help of PGA2. Then the non-repeating parts can be made as short as possible by including as much as possible in the repeating parts (by means of PGA4). After these transformations have been carried out, instruction sequence congruence now corresponds to syntactic equality.

3.2.2. Completeness of the program object equations

If the closed PGA expressions X and Y denote identical instruction sequences their equivalence can be demonstrated by means of the program object equations. A proof of this elementary fact is sketched below. It can be assumed that both expressions are in first canonical form. If X is finite, then Y is also finite and the proof of equality involves associativity of concatenation (PGA1) at most. Now let us assume that in both cases there is a repeating part. Then by means of PGA4 the two expressions can be rewritten in such a form that the non-repeating part is as short as possible. The non-repeating parts must be identical for both expressions. After these identical parts have been removed, a valid identity of the form $U^\omega = V^\omega$ must be inferred from the program object equations. Suppose that U has length k and that V has length l. Recall that $W^1 = W$ and $W^{n+1} = W; W^n$. Then U^l and V^k denote finite instruction sequence with equal lengths. Because both are initial segments of the same infinite instruction sequences they are equal and therefore the identity $U^l = V^k$ is provable by means of PGA1. Using PGA2 it follows that $U^\omega = (U^l)^\omega = (V^k)^\omega = V^\omega$, thus proving the identity that was looked for.

---

[12] This is the case if X contains no repetition and then $Y \equiv X$.

[13] We notice that $b; b^\omega$ is in first canonical form but b^ω is not.

4. Structural congruence equations

Next we present four equation schemes (equations parametrized by arbitrary natural numbers n, m and k each of which may be 0) which take care of the simplification of chained jumps. The schemes are termed PGA5-8, respectively. PGA8 is can be written as an equation by expanding X, but takes its most compact and readable form as a conditional equation. Program texts are considered structurally congruent if they can be proven equal by means of PGA1-8. Structural congruence of X and Y is indicated with $X =_{\mathrm{sc}} Y$, omitting the subscript if no confusion arises.

$\#n+1; u_1; \ldots; u_n; \#0 = \#0; u_1; \ldots; u_n; \#0$	(PGA5)
$\#n+1; u_1; \ldots; u_n; \#m = \#n+m+1; u_1; \ldots; u_n; \#m$	(PGA6)
$(\#n+k+1; u_1; \ldots; u_n)^\omega = (\#k; u_1; \ldots; u_n)^\omega$	(PGA7)
$X = u_1; \ldots; u_n; (v_1; \ldots; v_{m+1})^\omega \to$ $\#n+m+k+2; X = \#n+k+1; X$	(PGA8)

Examples of proofs of structural congruence:

- $\#10; +a; (-b; \#2)^\omega = \#8; +a; (-b; \#2)^\omega = \#6; +a; (-b; \#2)^\omega$
  $= \#4; +a; (-b; \#2)^\omega = \#2; +a; (-b; \#2)^\omega = \#2; +a; -b; (\#2; -b)^\omega$
  $= \#2; +a; -b; (\#0; -b)^\omega = \#2; +a; (-b; \#0)^\omega.$
- $a; (-b; \#2; -c; \#2)^\omega = a; -b; (\#2; -c; \#2; -b)^\omega = a; -b; (\#4; -c; \#2; -b)^\omega$
  $= a; -b; (\#0; -c; \#2; -b)^\omega = a; -b; \#0; -c; (\#2; -b; \#0; -c)^\omega$
  $= a; -b; \#0; -c; (\#0; -b; \#0; -c)^\omega = a; -b; (\#0; -c; \#0; -b)^\omega.$

Structural congruence allows a transformation into the second canonical form, which is introduced below, in Section 4.1. In the second canonical form inaction caused by a cycle of jumps occurs only in case #0 is used.

4.1. Canonical forms for PGA: the second canonical form

A program has (contains) a chained jump, if (perhaps after unfolding repetitions) it contains a subsequence of instructions of the following form:

$$\#n+1; u_1; \ldots; u_n; \#m.$$

The program object equations allow one to remove chained jumps by increasing the counter of the first jump (provided the second jump points to an instruction which is not a jump itself). If the second jump leads to a loop the first jump will do so as well and its counter can be set equal to 0.

We will consider an important subclass of the canonical terms, the so-called second canonical forms, for which the following two additional requirements hold:

(i) there are no chained jumps (i.e. PGA5-7 cannot be applied, even if intermediate applications of PGA4 are admitted[14]),

[14] For instance in order to bring a jump instruction in the front of a repeating subexpression as a preparation of an application of PGA7.

(ii) counters used for a jump into the repeating part of the expression are as short as possible (i.e. PGA7, for $k > 0$ and PGA8 cannot be applied).[15]

Every expression in first canonical form can be further transformed into the second canonical form. This transformation is sometimes useful as a preparation for further transformation. The transformation from the first canonical form into the second canonical form can be done using PGA4,5-8. Examples of transformations into the second canonical form:

- $a; \#9; (+b; !; c)^\omega =_{\mathrm{sc}} a; \#3; (+b; !; c)^\omega$,
- $-a; b; (+c; \#8; !)^\omega =_{\mathrm{sc}} -a; b; (+c; \#2; !)^\omega$.

Using PGA1-8 each PGA expression can be rewritten into a shortest structurally equivalent second canonical form.

4.2. Beyond structural congruence

Two programs can be considered equivalent also if they fail to be structurally congruent. Generating an exhaustive listing of such examples is not an easy task, however.

We will approach the general issue of program equivalence later under the heading of behavioral semantics (behavior extraction). At this point we can mention some plausible identities that escape structural congruence:

- $+a^\omega = a^\omega = -a^\omega$,
- $+a; \#1 = a; \#1 = -a; \#1$,
- $+a; u^\omega = a; u^\omega = -a; u^\omega$,
- $+a; !; ! = a; !; ! = -a; !; !$,
- $\#2; u; ! =!; u; !$

In the following section we will outline a notion of behavioral equivalence for programs. It is possible to identify programs as long as that identification does not imply that two programs with different behavior will have to be considered identical. The identities just mentioned have that property.

5. Program behaviors with inaction

Prior to an explanation of behavioral equivalence of programs, a description of program behaviors is useful. Evident though this may seem, a remarkable explanatory paradox occurs:

(i) there are a number of substantially different definitions of the behavior of a program, none of them being 'the best',

(ii) in order to obtain a comprehensible explanation at least one definition of behavior must be presented in sufficient detail.

We have chosen to cover the details of BPPA (basic polarized process algebra[16]), admitting beforehand that this is just one of several possibilities to go ahead. BPPA has the advantage of not standing on the feet of a substantial body of other theory. (This may prove to be a disadvantage as well when it comes to more involved aspects of the theory.)

[15] All programs without chained jumps and without repetition operator are also in second canonical form.

[16] This jargon merits some explanation: 'basic' refers to the absence of parallelism or concurrency; 'polarized' refers to the fact that all actions involved have a polarity, in this case the actions serve as requests for an environment which will reply with a change of state and a boolean return value; 'process' is just another word for behavior; 'algebra' refers to the description based on constants and operators.

5.1. Primitives of BPPA

BPPA is based on a collection Σ of basic instructions. Because we are discussing behaviors instead of programs now these basic instructions will also be called actions, which is customary in process theory. Each basic instruction is supposed to produce a boolean value when executed. Further BPPA has two constants and two composition mechanisms. In addition there is a family of approximation operators. The constants are meant to model termination and inaction. For a collection Σ of actions BPPA_Σ denotes its associated family of program behaviors.

Termination. With S (stop) terminating behavior is denoted; it does no more than terminate.

Inactive behavior. By D (inaction or sometimes just 'loop') an inactive behavior is indicated. It is a behavior that represents the impossibility of making real progress, for instance an internal cycle of activity without any external effect whatsoever.[17]

S and D are contained in BPPA_Σ for each Σ. The composition mechanisms are postconditional composition and action prefix. Action prefixing is merely an abbreviation for an instance of postconditional composition.

Postconditional composition. For action $a \in \Sigma$ and behaviors P and Q in BPPA_Σ

$$P \trianglelefteq a \trianglerighteq Q$$

denotes the behavior in BPPA_Σ that first performs a and then either proceeds with P if `true` was produced or with Q otherwise.

Action prefix. For $a \in \Sigma$ and behavior $P \in \mathrm{BPPA}_\Sigma$

$$a \circ P = P \trianglelefteq a \trianglerighteq P.$$

5.2. Approximation of program behaviors

Program behaviors can be finite and infinite. A behavior is called finite if there is a finite upper bound to the number of consecutive basic instruction behaviors (=actions) it can perform.[18] All finite behaviors are made from S and D by means of a finite number of applications of postconditional composition. The definition of infinite behaviors makes use of so-called projective sequences. These in turn require the approximation operators π_n for n a natural number ($n \in NN$). On finite behaviors the approximation operators are determined by these equations:

$$\begin{aligned}
&\pi_0(P) = D,\\
&\pi_{n+1}(S) = S,\\
&\pi_{n+1}(D) = D,\\
&\pi_{n+1}(P \trianglelefteq a \trianglerighteq Q) = \pi_n(P) \trianglelefteq a \trianglerighteq \pi_n(Q).
\end{aligned}$$

A projective sequence is a sequence $(P_n)_{n\in NN}$ such that for each $n \in NN$ $\pi_n(P_{n+1}) = P_n$. Projective sequences are considered equal exactly if all components are equal. Projective sequences can be used to represent finite as well as infinite behaviors. A finite behavior P is represented as the sequence $(\pi_n(P))_{n\in NN}$. Postconditional composition (and

[17] Inaction typically occurs in case an infinite number of consecutive jumps is performed; for instance $(\#1)^\omega$.

[18] As a consequence D is considered a finite behavior because it produces zero actions.

action prefix at the same time) is defined on infinite behaviors (i.e. on projective sequences) as follows: let $P = (P_n)_{n\in NN}$ and $Q = (Q_n)_{n\in NN}$, then $P \trianglelefteq a \trianglerighteq Q = (R_n)_{n\in NN}$ with $R_0 = D$ and $R_{n+1} = P_n \trianglelefteq a \trianglerighteq Q_n$. One proves the sequence $(R_n)_{n\in NN}$ to be a projective sequence with induction on n.

Summarizing these definitions: program behaviors are projective sequences of finite program behaviors. Finite program behaviors are built from the constants S and D using postconditional composition. Action prefix serves only as an abbreviation. Examples:

- $\pi_2(b \circ D) = b \circ D$,
- $\pi_3((b \circ S) \trianglelefteq c \trianglerighteq (e \circ e \circ f \circ S)) = (b \circ S) \trianglelefteq c \trianglerighteq (e \circ e \circ D)$,
- $\pi_4((c \circ b \circ D) \trianglelefteq c \trianglerighteq (e \circ e \circ f \circ f \circ D)) = (c \circ b \circ D) \trianglelefteq c \trianglerighteq (e \circ e \circ f \circ D)$.

Equality of infinite behaviors can easily be retrieved from equality of finite behaviors. Two (finite or infinite) behaviors are equal exactly if for each natural number n, the nth approximations of the two behaviors are equal. It follows that once it has been defined (or understood) when two finite behaviors are equal, that definition automatically extends to the case of infinite behaviors.

Finite approximations of behaviors are considered equal if and only if they have exactly the same form (using the notation suggested above).[19]

5.3. Solving equations in BPPA

The world of BPPA admits the solution of many equations and systems of equations. Without embarking on an exposition of the general theory of equation solving over BPPA some examples are still useful. Consider the equation:

$$E(P): \quad P = (S \trianglelefteq a \trianglerighteq D) \trianglelefteq b \trianglerighteq P.$$

A solution $P = (P_n)_{n\in NN}$ of $E(P)$ is given by $P_0 = D$, $P_1 = D \trianglelefteq b \trianglerighteq D$, $P_2 = (D \trianglelefteq a \trianglerighteq D) \trianglelefteq b \trianglerighteq (D \trianglelefteq b \trianglerighteq D)$, and for $n > 0$, $P_{n+2} = (S \trianglelefteq a \trianglerighteq D) \trianglelefteq b \trianglerighteq P_{n+1}$. It is easily demonstrated by means of induction on n that each of the approximations of P is unique. Therefore the equation $E(P)$ has a unique solution in BPPA.

As a second example consider the following system of two equations

$$E(P, Q): \quad P = (S \trianglelefteq a \trianglerighteq Q) \trianglelefteq b \trianglerighteq P, \quad Q = (P \trianglelefteq c \trianglerighteq Q) \trianglelefteq d \trianglerighteq D.$$

The following pair of projective sequences constitutes a unique solution for $E(P, Q)$.

$$\begin{aligned} &P_0 = Q_0 = D,\\ &P_{n+1} = \pi_{n+1}((S \trianglelefteq a \trianglerighteq Q_n) \trianglelefteq b \trianglerighteq P_n),\\ &Q_{n+1} = \pi_{n+1}((P_n \trianglelefteq c \trianglerighteq Q_n) \trianglelefteq d \trianglerighteq D). \end{aligned}$$

5.4. Behavior extraction equations

Semantic equations will describe the behavior of complex programs in terms of the behavior of their constituent parts. The behavior extraction operator $|-|$ assigns a behavior to a program object. Structurally congruent program objects will be assigned identical behaviors. Behavioral equivalence will not be compositional. We will comment on compositionality in Section 5.5.1.

[19] Our presentation of projective sequences is based on [1]. The process algebra ACP defined in that paper cannot be used here, however, because ACP fails to contain a convincing counterpart of postconditional composition.

In combination with an intuitive understanding of a behavior, of the composition operators described for behaviors and of the irreducible behaviors, the semantic equations provide memorizable detail for the intuitions given for the building blocks of PGA. By memorizing (and understanding) the semantic equations one may reasonably claim to know the semantics of PGA. Of course, it should be kept in mind that the semantic equations below provide only one out of a number of semantic options. In order to fully understand these equations, one should notice that there are no overlaps leading to ambiguities, and that each closed program expression will match one of these expressions. One may view the equations as defining equations for the behavior extraction operator. This definition uses induction on the number of instructions in a program and, on top of that, an induction on the counter of a jump instruction.[20]

In the equations below a ranges over the basic instructions in Σ, u ranges over all primitive instructions and X ranges over arbitrary program objects.

5.4.1. Restricting attention to infinite instructions sequences

Behavior extraction is best defined on infinite program objects. For finite program objects the definition then reads

$$|X| = |X; (\#0)^{\omega}|.$$

This definition expresses the idea that a 'missing' instruction leads to inaction.

5.4.2. Semantic equations for void basic instructions and tests

The behavior $|X|$ of (infinite) program object X is determined using a complex tail recursion. In all cases the behavior starts with the behavior of the first primitive instruction, if possible

$$|!; X| = S,$$
$$|a; X| = |X| \trianglelefteq a \trianglerighteq |X|,$$
$$|+a; u; X| = |u; X| \trianglelefteq a \trianglerighteq |X|,$$
$$|-a; u; X| = |X| \trianglelefteq a \trianglerighteq |u; X|.$$

5.4.3. Semantic equations for the jump instructions

The case of the jump instructions requires a case distinction on the counter of the jump. In case that counter is zero, inaction will occur. In case that counter is one, it skips itself. In case the counter exceeds one, the first of the subsequent instructions is dropped and the counter decreases accordingly

$$|\#0; X| = D,$$
$$|\#1; X| = |X|,$$
$$|\#k + 2; u; X| = |\#k + 1; X|.$$

5.4.4. Inaction and non-trivial loops

The above equations should be used to obtain successive steps of the behavior of a program object X. The equations may be applied infinitely often without ever generating the

[20] The design of PGA has been 'optimized' as to make the behavior extraction operator definition as simple as possible. Abstract machines are avoided entirely.

behavior of a single basic instruction. In that case the program has a non-trivial loop and its behavior will be identified with D. By doing this we obtain for instance: $|(\#1)^\omega| = D$ and $|b; (\#2; a)^\omega| = b \circ D$. (We notice that these facts can be derived with the help of PGA5-8.)

This matter can be formulated more sharply as follows: if (for behavior $|X|$) the behavior extraction equations fail to prove $|X| = S$ or $\pi_1(|X|) = a \circ D$ for some $a \in \Sigma$ then $|Y| = D$ for some Y with $Y =_{\mathrm{sc}} X$.

5.4.5. Repetition generates infinite behaviors

If X has no repetition $|X|$ is a finite behavior. Programs with repetition can have infinite behaviors.

As an example consider $X = a^\omega$. We find $|X| = a \circ a \circ a \circ \cdots$. Using projective sequence notation: $|X| = (P_n)_{n \in NN}$ with $P_0 = D$, $P_1 = a \circ D$, $P_2 = a \circ a \circ D$, etc.

5.4.6. Examples

For a better understanding of the implications of the above semantic equations we provide some examples: $|a| = a \circ D\ (= D \trianglelefteq a \trianglerighteq D)$, $|a; !| = a \circ S$, $|\#1; !| = S$, $|+a; !| = S \trianglelefteq a \trianglerighteq D$, $|\#2; !; !| = S$, $|a^\omega| = a \circ |a^\omega|$, and $|+a; -b; c; !| = (S \trianglelefteq b \trianglerighteq (c \circ S)) \trianglelefteq a \trianglerighteq c \circ S$.

5.5. Behavioral equivalence

Two programs X and Y are behaviorally equivalent (denoted by or $X \equiv_{\mathrm{be}} Y$) if $|X| = |Y|$. This in turn holds precisely if for all $n \in NN$, $\pi_n(|X|) = \pi_n(|Y|)$. It can be shown that it is decidable whether or not $X \equiv_{\mathrm{be}} Y$ for closed program algebra expressions X and Y. As an example we mention: $a; b; c \equiv_{\mathrm{be}} a; \#2; \#1; b; c$. The proof reads as follows:

$$
\begin{aligned}
&|a; \#2; \#1; b; c| \\
&\quad = a \circ |\#2; \#1; b; c| \\
&\quad = a \circ |\#1; b; c| \\
&\quad = a \circ |b; c| \\
&\quad = |a; b; c|.
\end{aligned}
$$

A second example: $+a; !; ! \equiv_{\mathrm{be}} a; !; \#1$, because $|+a; !; !| = |!; !| \trianglelefteq a \trianglerighteq |!| = S \trianglelefteq a \trianglerighteq S = a \circ S$ and $|a; !; \#1| = a \circ |!; \#1| = a \circ S$.

5.5.1. Non-compositionality of behavioral equivalence

Non-compositionality of behavioral equivalence follows from a simple counterexample: consider $X = \#2; a; b; !$ and $Y = \#2; c; b; !$. It will be clear that $|X| = |Y| = b \circ S$. On the other hand $|\#2; X| = a \circ b \circ S$, that differs from $|\#2; Y| = c \circ b \circ S$.

Non-compositionality is just another way of expressing that an equivalence fails to be a congruence. So behavioral equivalence is not a congruence.

5.6. Behavioral congruence

Behavioral congruence is the largest congruence relation contained in behavioral equivalence. Behavioral congruence of X and Y is denoted with $X =_{\mathrm{bc}} Y$. It can be shown that

$X =_{\mathrm{bc}} Y$ if for all $n, m \in NN$, $\#n; X; !^m =_{\mathrm{be}} \#n; Y; !^m$.[21] Structural congruence implies behavioral congruence but not conversely. Some examples: $!; a =_{\mathrm{be}} !$ and $!; a \neq_{\mathrm{bc}} !$, further $+a; !; ! =_{\mathrm{bc}} -a; !; !$ but $+a; !; ! \neq_{\mathrm{sc}} -a; !; !$.

5.6.1. Long jumps are necessary

For each natural number n a program object X can be found such that for no PGA expression Y making use of jump instructions $\#k$ with $k < n$ only: $X =_{\mathrm{be}} Y$.

For denoting an X the following set of basic instructions will be used: $\Sigma = \{a\} \cup \{b_1, b_2, b_3, \ldots\}$.

An example is as follows: let for each n:

$Q_n = (+a; \#2n + 2)^{n+1}$,

$R_n = (b_1; \#2n + 1; b_2; \#2n + 1; \ldots; b_n; \#2n + 1; b_{n+1}; \#2n + 1)^{\omega}$, and

$P_n = Q_n; !; R_n$.

The semantics is clear from an example

$$|P_2| = |b_1^{\omega}| \trianglelefteq a \trianglerighteq (|b_2^{\omega}| \trianglelefteq a \trianglerighteq (|b_3^{\omega}| \trianglelefteq a \trianglerighteq S)).$$

Suppose that $|P_n| = |Y|$, where all jumps in Y have counter below n, a contradiction will be derived. Let $Y = u_1 \ldots; u_m; (v_1; \ldots; v_k)^{\omega}$. The instruction sequence for Y can be unfolded by writing $u_{m+i} = v_{i \bmod k}$. Let $Y_t = u_t; u_{t+1}; \ldots$ for all t.

Let $V_i = \{j > m | Y_j \equiv_{\mathrm{be}} b_i^{\omega}\}$ for $1 \leqslant i \leqslant n + 1$. All V_i are infinite, pairwise disjoint and have gaps (number of consecutive non-V_i elements between two members of V_i) with a size less than n. (Larger gaps come from jumps, but a jump with counter t gives rise to a gap of at most size $t - 1$.)

Choose i, p and q such that $m < p < q$ and $p, q \in V_i$ and $q - p$ maximal under the constraint that no r between p and q is in V_i. Thus the pair p, q constitutes a maximal gap within the unfolded repeating part of the instruction sequence. If $q - p \geqslant n$, a contradiction is found with the assumptions for all V_j. Now assume that $q - p < n$. Then at least one of the V_j (say V_r with $r \neq i$) has no element between p and q.

Moving one cycle of the iteration further $p + k$ and $q + k$ constitute a gap with equivalent properties. Now V_r must have an element s with $n < s < p + k$. Let s' be maximal among such s, then s' is the first member of a gap for V_r exceeding $q - p$, thus reaching a contradiction.

6. Program behavior and input–output relations

In some cases it is useful to see a program as denoting a transformation from input values into output values. Though conceptually simpler this is in fact a more abstract view. Input–output transformations can be derived from program behaviors rather than from a program itself. Assigning an input–output mapping to a behavior rests on several conceptual prerequisites. A state space is needed with elements of the state space playing the role of inputs as well as of outputs. Every basic instruction must be viewed as a transformation of the states in the state space, producing a boolean whenever applied. The change of state resulting from the operation of a basic instruction is given by the so-called effect operation. The actions of a behavior are to be applied consecutively to the state taken as an input value.

[21] Here $Z; V^0 = Z$ and $Z; V^{k+1} = Z; V; V^k$.

The state reached after a final action has been performed then represents the output value of a computation. These intuitions are put in a formal form below:

For a given Σ we define the notion of a deterministic state space as follows. A deterministic state space consists of a set V together with functions $\mathit{effect}_a(-) : V \to V$ for each action $a \in \Sigma$,[22] and a function $y_a(-) : V \to \{\mathtt{true}, \mathtt{false}\}$ for each $a \in \Sigma$ (the boolean yield function) which determines the boolean reply $y_a(v)$ produced when instruction a is performed in a state v in V. A behavior P determines a function of type $V \to V \cup \{D\}$, according to the equations below. Here D is an object (state) rather than a behavior, now representing a default value. D represents a value that can't be computed. We will write $P \bullet v$ for the value of this function.[23] It represents what P computes on input v in V. The definition of '$\bullet$' is with induction on the top-level structure of P

(1) $D \bullet v = D$,
(2) $S \bullet v = v$,
(3) $(a \circ P) \bullet v = P \bullet \mathrm{effect}_a(v)$,
(4) $(P \trianglelefteq a \trianglerighteq Q) \bullet v = (a \circ P) \bullet v \ \mathtt{if}\ y_a(v), \ \mathtt{otherwise}\ (a \circ Q) \bullet v$
$(= ((a \circ P) \bullet v) \lhd y_a(v) \rhd ((a \circ Q) \bullet v)$, using the conditional operator notation of [4]).

Whenever these identities do not determine a value for $|X| \bullet v$, the cause must be that the generated computation proceeds indefinitely. Then we take $|X| \bullet v = D$ in order to express that the computation produces no result. In other words: $|X| \bullet v = D$ precisely if for all $n \in NN$, $\pi_n(|X|) \bullet v = D$. For a given state space with effect functions and tests F, one obtains an equivalence relation $\equiv_{ioF}$ on program objects which declares X and Y equivalent whenever the corresponding mappings $|X| \bullet v$ and $|Y| \bullet v$ coincide on all v in V.

7. Program algebra projections

A program algebra projection is a mapping from a set L into SPI_Σ for an appropriate action set Σ. Such a mapping can be defined with many techniques. Evidently a program algebra projection φ turns objects in L into programs. It follows that using φ an algorithmic or operational meaning can be assigned to objects in L. One may write $|X|_L = |\varphi(X)|$ for X in L.

It may well be the case that a far more elegant way of assigning $|X|$ to X can be found than the detour via $|\varphi(X)|$. However, the detour is acceptable, and it does not commit one to any claim concerning the existence of high-level or denotational methods to obtain a behavioral semantics for objects in L.

7.1. Programming languages

A programming language can be defined as a pair (L, φ) with L some collection of textual objects and φ a program algebra projection. PGA expressions are trivially viewed as instruction sequences, thus allowing to view PGA as a program notation. In some cases, it may be necessary (and is considered acceptable) to allow the use of auxiliary actions

[22] In some applications (though not in this paper) the notation $v_{\partial a}$ is used for $\mathit{effect}_a(v)$, as it is a more compact notation.

[23] A more specific notation for this operator $(P \bullet v)$ is $P \bullet^{io} v$.

in the notation PGA into which the projection translates the objects in L. The most obvious example is the use of stack manipulation actions needed if L contains instructions manipulating nested expressions for mathematical values.

It is reasonable if not mandatory to require that the program algebra projection is a computable mapping. This definition is to be read as a criterion rather than as a dogmatically limiting constraint. The present definition of a programming language is simplistic in nature but, at least in principle, it covers such illustrious examples as COBOL, Java, C, and C++, and other programming paradigms such as ASF+SDF, PROLOG and CLEAN.

The projection can be seen as a theoretical compiler, optimized for human understanding rather than for one of the more conventional objectives, such as: code-compactness (small expression that represents the output of the projection), target code space and/or time efficiency, and compiler execution space and/or time efficiency.

7.2. A program notation for PGA: PGLA

PGLA is a program notation specifically designed for representing programs for which a canonical form for PGA is known. Moreover, it has the advantage of allowing each of its programs to be denoted by means of a finite sequence of instructions. To that end an additional instruction is introduced: the repeat instruction $\backslash\backslash\#n$, for any natural number $n > 0$. A program text ending with $\backslash\backslash\#n$ will repeat its last n instructions, excluding the repetition instruction itself. Instructions to the right of a repetition instruction are irrelevant and can be deleted. If a repetition instruction ends an instruction sequence shorter than its counter, that list will be padded with an initial sequence of instructions #0, in order to have a list of instructions of sufficient length. PGLA programs have the form $u_1; \ldots; u_k$, with each u_i either a primitive instruction of PGA or a repeat instruction.

The program algebra projection `pgla2pga` works as follows:

(i) for a program text without repeat instruction, its projection is the instruction sequence denoted by that text viewed as a PGA expression,

(ii) for a program containing a repeat instruction all instructions following the first (i.e. left-most) repeat instruction are removed; subsequently one of the following rules is applied (assuming that the u_i are primitive PGA instructions):

$$\text{if } k > n \text{ then } \texttt{pgla2pga}(u_1; \ldots; u_k; \backslash\backslash\#n) = u_1; \ldots; u_{k-n}; (u_{k-n+1}; \ldots; u_k)^\omega,$$
$$\text{if } k = n \text{ then } \texttt{pgla2pga}(u_1; \ldots; u_k; \backslash\backslash\#n) = (u_1; \ldots; u_k)^\omega,$$
$$\text{if } k < n \text{ then } \texttt{pgla2pga}(u_1; \ldots; u_k; \backslash\backslash\#n) = u_1; \ldots; u_k; ((\#0)^{n-k}; u_1; \ldots; u_k)^\omega.$$

Examples:

$$\texttt{pgla2pga}(a; \#3; \backslash\backslash\#2) = (a; \#3)^\omega,$$
$$\texttt{pgla2pga}(b; a; \#3; \backslash\backslash\#2) = b; (a; \#3)^\omega,$$
$$\texttt{pgla2pga}(a; \#3; \backslash\backslash\#3) = a; \#3; (\#0; a; \#3)^\omega,$$
$$\texttt{pgla2pga}(a; \#3; \backslash\backslash\#4) = a; \#3; (\#0; \#0; a; \#3)^\omega.$$

We will write $|X|_{pgla} = |\texttt{pgla2pga}(X)|$.

PGLA being available, it is obvious that a program algebra projection for a set F can be represented by a mapping `f2pgla`, which maps F texts into PGLA texts. Below program algebra projections will be provided via translations into PGLA. Such translations may take several steps through intermediate languages.

7.3. Forward and backward jumps in PGLB

PGLB is an important variation on PGLA. It is important because it is much closer than PGLA to program notations that are used in practice. It has an additional backward jump instruction written $\backslash\#k$. In the presence of backward jumps, repetition becomes a redundant feature and is left out. Each program is a sequence $u_1; \ldots; u_k$ of instructions.

The program algebra projection `pglb2pgla` is defined using auxiliary operations ψ_j. Given PGLB program $u_1; \ldots; u_k$ we write

$$\texttt{pglb2pgla}(u_1; \ldots; u_k) = \psi_1(u_1); \ldots; \psi_k(u_k); \#0; \#0; \backslash\backslash\#k + 2.$$

The operations ψ_j (for $1 \leqslant j \leqslant k$) serve to translate the individual program instructions. This translation is position dependent, the position being taken into account in the form of a subscript to the operator. The defining rules for these operations then read as follows:

$$\begin{aligned}
&\psi_j(\#l) = \#l \ \texttt{ if } j + l \leqslant k,\\
&\psi_j(\#l) = \#0 \ \texttt{ if } j + l > k,\\
&\psi_j(\backslash\#l) = \#k + 2 - l \ \texttt{ if } l < j,\\
&\psi_j(\backslash\#l) = \#0 \ \texttt{if } l \geqslant j,\\
&\psi_j(u) = u \ \texttt{ otherwise}.
\end{aligned}$$

Examples:

$$\begin{aligned}
&\texttt{pglb2pgla}(+a) = +a; \#0; \#0; \backslash\backslash\#3,\\
&\texttt{pglb2pgla}(+a; !; \backslash\#2; \#5; -b; !) = +a; !; \#6; \#0; -b; !; \#0; \#0; \backslash\backslash\#8.
\end{aligned}$$

The extension of $u_1; \ldots; u_k$ with ; #0; #0 is needed in order to prevent the projected program from proceeding correctly if the program before projection fails to end with a termination instruction. The idea of the projection is that backward jumps can be replaced by forward jumps if the entire program is repeated.

The projection from PGLB into PGA is defined by

$$\texttt{pglb2pga}(X) = \texttt{pgla2pga}(\texttt{pglb2pgla}(X))$$

and satisfies

$$\texttt{pglb2pga}(u_1; \ldots; u_k) = (\psi_1(u_1); \ldots; \psi_k(u_k); \#0; \#0)^{\omega},$$

where the operations ψ_j (for $1 \leqslant j \leqslant k$) are defined as above.

7.4. Conventional termination in PGLC

PGLC is a minor variation on PGLB. PGLC has no explicit termination instruction. Termination takes place when the last action in the list has been executed (and was no backward jump) or when a forward or backward jump is made to an instruction outside the list. The importance of PGLC is that it is closer (than PGLB) to the termination conventions used in existing program notations of similar expressive power, in particular various assembly languages. The program algebra projection `pglc2pglb` is given by

$$\texttt{pglc2pglb}(u_1; \ldots; u_k) = \psi_1(u_1); \ldots; \psi_k(u_k); !; !$$

The auxiliary operators ψ_j are given by

$$\psi_j(\#l) = !\ \texttt{if}\ \ j + l > k,$$
$$\psi_j(\backslash\#l) = !\ \texttt{if}\ \ l \geqslant j,$$
$$\psi_j(u) = u\ \ \texttt{otherwise}.$$

Examples:

$$\texttt{pglc2pglb}(+b) = +b; !; !,$$
$$\texttt{pglc2pglb}(+c; \#10; \backslash\#1; -c; \#2; +b) = +c; !; \backslash\#1; -c; !; +b; !; !.$$

7.5. Absolute jumps in PGLD

PGLD is yet another variation on PGLA. As in PGLC, repetition and explicit termination have been omitted. The termination conventions are the same as in PGLC. Instead of forward and backward jumps PGLD allows absolute jumps; $\#\#k$ for a natural number k, is an instruction which makes a control move to (the beginning of) the kth instruction of the program. The importance of PGLD is that (like PGLC) it is close to existing conventions but quite different in style. If $k = 0$ termination will occur. We find a program algebra projection for PGLD by first projecting it into PGLC, and subsequently using the projection for PGLC.

The program algebra projection `pgld2pglc` is obtained as follows. Given PGLD program $u_1; \ldots; u_k$ we define

$$\texttt{pgld2pglc}(u_1; \ldots; u_k) = \psi_1(u_1); \ldots; \psi_k(u_k).$$

The operations ψ_j then read as follows:

$$\psi_j(\#\#l) = \#l - j\ \ \texttt{if}\ l \geqslant j,$$
$$\psi_j(\#\#l) = \backslash\#j - l\ \ \texttt{if}\ l < j,$$
$$\psi_j(u) = u\ \ \texttt{otherwise}.$$

Example:

$$\texttt{pgld2pglc}(a; +b; \#\#1; \#\#8; c; \#\#5; f) = a; +b; \backslash\#2; \#4; c; \backslash\#1; f.$$

7.6. Labels and goto's in PGLDg

Based on PGLD we will outline further language extensions (or modifications) which allow one to use labels. In PGLDg goto's and labels are introduced. A label is just a natural number. PGLDg omits the absolute jumps as those can be replaced by an appropriate use of labels. The meaning of PGLDg is given in terms of a projection back to PGLD.

7.6.1. Program expression syntax for PGLDg

As in PGLD, the only composition mechanism of programs in PGLDg is concatenation. PGLDg differs from PGLD in containing labels and goto's. Because they can be emulated by means of labels and goto's, absolute jumps have been deleted, the language becoming more homogeneous as a result. Incorporation in PGLDg of absolute jumps would not introduce any conceptual or semantic difficulties, however, and can under circumstances be quite practical.

As labels we will use the natural numbers. £k denotes a label with name k with ##£k a jump towards a label is denoted. The effect of the introduction of labels on the collection of instructions is hardly significant. We outline the class of instructions below:

- Actions and tests as in PGLD. These comprise: void basic instruction (a), positive test instruction ($+a$), negative test instruction ($-a$),
- the termination instruction (!), only (re)introduced here because of its ease of use, in view of the fact that PGLDg can play a practical role in program theory.
- Label catch instruction. The instruction £k, for k a natural number, represents a visible label. As an action it is a skip in the sense that it will not have any effects on a state space. (A label catch instruction is called a C-occurrence of the label.)
- Absolute goto instruction. For each natural number k the instruction ##£k represents a jump to the (beginning of) the first (i.e. the left-most) label catch instruction in the program which is labeled by the label k. If no such instruction can be found termination of the program execution will occur. (An absolute goto instruction is called a G-occurrence of its label.)

An example of a PGLDg program is

£0; $-a$; ##£1; ##£0; £1.

In this program a is repeated until it returns value `false`. That is also the functionality of the simpler program £0; $+a$; ##£0.

7.6.2. A projection from PGLDg to PGLD

We take it for granted that the reader has an intuitive grasp of the meaning of PGLDg programs. Nevertheless we will determine their meaning by providing the details of a projection from PGLDg into PGLD leading to the following semantic equation:

$$|X|_{\mathrm{pgldg}} = |\mathtt{pgldg2pgld}(X)|_{\mathrm{pgld}}.$$

The projection `pgldg2pgld` from PGLDg to PGLD is quite simple. For a program X, the following steps have to be taken:

(1) Replace for each label k, the goto instruction ##£k by ##n with n equal to the smallest j such that the jth instruction contains an C-occurrence of the label k, if that instruction exists, or replace it by ##0 otherwise.
(2) Each label catch instruction £k is replaced by ##$j+1$ with j the instruction number of that label catch instruction. (This is a slightly cumbersome way to produce a skip instruction.)

More formally the projection `pgldg2pgld` works as follows on a program $u_1; \ldots; u_k$:

$$\mathtt{pgldg2pgld}(u_1; \ldots; u_k) = \psi_1(u_1); \ldots; \psi_k(u_k),$$

where the auxiliary operation ψ_j reads as follows:

$$\begin{aligned}
&\psi_j(!) = \#\#0,\\
&\psi_j(\#\#£k) = \#\#\mathtt{target}(k),\\
&\psi_j(£k) = \#\#j+1,\\
&\psi_j(u) = u \quad \mathtt{otherwise}.
\end{aligned}$$

The auxiliary function `target`(k) produces for k the smallest number j such that the jth instruction of the program is of the form £k, if such a number exists and 0 otherwise.

Example:

$$\texttt{pgldg2pgld}(£0; +a; \#\#£1; \#\#£0; £1) = \#\#2; +a; \#\#5; \#\#1; \#\#6.$$

7.7. Preparing for instruction refinement in PGLE

PGLE is a sublanguage of PGLDg. The restriction posed on PGLDg programs justifying their classification in PGLE is this: each test instruction (positive or negative) must always be immediately followed by a goto instruction, or a termination instruction. The projection of PGLE into PGLDg is trivial: nothing needs to be changed.

The advantage of PGLE over PGLDg emerges when projections need instruction refinement. Instruction refinement takes place if as part of a projection an instruction u is replaced by a sequence of instructions $u_1; u_2; u_3; \ldots$.

To see the problem with PGLDg consider the program $+a; u; c$, where u is some 'advanced' (newly introduced) instruction, in need for projection semantics. Assume further that a projection e.g. translating this program into a program not involving u needs to split (refine) u into u_1, u_2; then the result $+a; u_1; u_2; c$ is problematic because after a failing test a the instruction u_2 is performed, whereas $+a; u; c$ will perform the instruction c in case the preceding instruction a has returned `false`.

Instructions that can be added to PGLE and be removed using a projection operator applying instruction refinements, will be called advanced control instructions. By extending PGLE with advanced control instructions and finding appropriate projections back to PGLE a significant number of program notations can be developed.

8. Conditional constructs and while loops

On top of PGLE it is easy and, for the job of denoting programs clearly helpful, to offer the conventional conditional constructs. These can be viewed as design patterns from the point of view of PGLE programming. We will now provide a syntax PGLEc allowing conditional instructions. Then a projection is given from PGLEc into PGLE.

8.1. Syntax for conditional instructions

The syntax for conditional constructs takes the form of three new (advanced control) instructions:

Conditional instruction. For a basic instruction $a \in \Sigma$ the instructions $+a\{$ and $-a\{$ initiate the text of a conditional construct.

Then/else separator. The instruction '$\}\{$' connects two program sections that are enclosed in braces.

End brace. The instruction '$\}$' serves as a closing brace in connection with its complementary opening brace.

An example clarifies the intended meaning of (this rendering of) conditional instructions. The subsequent program algebra projection into PGLE formalizes the same. The program

$$+a\{; b; +c; \#\#£0; \#\#£1; \}\{; e; f; £0; g; £1; h; \}; -a; !; b$$

will start with the execution of a. If that yields `true` the first branch is taken. Execution of the first branch will unavoidably lead to a jump into the second branch. It depends on

the yield of c at which position in the second branch the computation proceeds. If the yield of the action a mentioned before is `false` then the second branch (beginning just after the separator instruction '}{') is executed.

8.2. Projection semantics for PGLEc

We will anticipate on the projection semantics of PGLEc programs by providing an intermediate program notation PGLEca. PGLDca provides quantitative information concerning the bracketing structure. This information shows up in annotated instructions. Instead of the instructions '}{' and '}' the following instructions are used in PGLEca:

Annotated then/else separator. For any natural number n the instruction '}n{' connects two program sections that are enclosed in braces. The annotation n indicates that the closing brace matches an opening brace occurring in instruction number n. If $n = 0$ the annotation indicates the absence of a matching opening brace.

Annotated end brace. The instruction '}n' serves as a closing brace in connection with its complementary opening brace. The annotation n indicates that the matching opening brace is found in the nth instruction. If $n = 0$ this indicates the absence of a matching opening brace in the program.

8.2.1. Projecting PGLEc to PGLEca

The projection `pglec2pgleca` works as follows on $X = u_1; \ldots; u_l$: replace each instruction containing a closing brace by an annotated corresponding instruction in such a way that the resulting program has correct annotations. This can be done in a unique way. Suppose the instruction u_k needs to be augmented with an annotation. The annotation is found by working backwards (from the kth position) in the program and maintaining an integer count of observed braces starting with 0. Moving to the left instruction by instruction (beginning with instruction u_k), the counter is increased whenever an opening brace is found and it is decreased whenever a closing brace is found. If the count becomes positive for the first time, say when processing instruction u_n, n is the required annotation for u_k. If u_k is } it is replaced by }n; if it is }{, its annotated version is }n{. If that fails to happen, the annotation becomes 0. A PGLEc program translating into a PGLEca program involving 0 annotations is considered syntactically correct.

8.2.2. A projection operator for PGLEca

The projection `pgleca2pgle` determines the semantics of PGLEca programs and after composition with `pglec2pgleca` it determines behavior extraction for PGLEc. It simultaneously removes all occurrences of positive and negative conditional instructions, as well as the instructions containing annotated closing braces. All labels that occur in the program before transformation are increased by $k + 1$, with k the number of instructions of the program. This will guarantee that labels of the form i with $i \leqslant k$ are new and cannot interfere with existing labels. Let $X = u_1; \ldots; u_k$. Then we read

$$\texttt{pgleca2pgle}(X) = \psi_1(u_1); \ldots; \psi_k(u_k)$$

with the auxiliary operations ψ_i determined by the following rewrite rules:

$$\psi_i(\#\#\pounds l) = \#\#\pounds l + k + 1,$$
$$\psi_i(\pounds l) = \pounds l + k + 1,$$
$$\psi_i(+a\{) = -a; \#\#\pounds i,$$

$$\psi_i(-a\{) = +a; \#\#£i,$$
$$\psi_i(\}n) = £n,$$
$$\psi_i(\}n\{) = \#\#£i; £n,$$
$$\psi_i(u) = u \quad \texttt{otherwise}.$$

Example:

$$\texttt{pglec2pgle}(+a\{; b; \}\{; c; \}; \#\#£0; d; £0)$$
$$= -a; \#\#£1; b; \#\#£3; £1; c; £3; \#\#£9; d; £9.$$

8.3. Advanced control instructions for a while loop

Structured iteration constitutes an important part of structured programming. By adding four new instructions (named the positive and the negative while-loop header, the unconditional while-loop header, and the end of while-loop) to the arsenal of PGLEc, PGLEcw is obtained as follows:

Positive/negative while-loop header. For an action $a \in \Sigma$ the instructions $+a\{*$ and $-a\{*$ initiate the text of a while loop.

Unconditional while-loop header. The instruction $\{*$ initiates the text of an unconditional while-loop.

End of while-loop. The instruction $*\}$ marks the end of the body of a while-loop.

In order to provide a projection semantics to PGLEcw an extension of PGLEca to PGLEcwa is needed. The new PGLEcwa instructions are $*\}n$ (for non-zero $n \in NN$) indicating a closing brace corresponding to an opening brace contained in a while-loop header instruction at position n), $+a\{*n$ and $-a\{*n$ (indicating the header of a while-loop having its closing brace at position n if there is a closing brace in the program, n being set to 0 otherwise), and $\{*n$ (indicating an unconditional while-loop header having its closing brace at instruction n).

8.4. Projection semantics for PGLEcw

The projection `pglecw2pglecwa` is obvious. The projection `pglecwa2pgle` works just like that of `pgleca2pgle` with the addition of the following clauses in the definition of the operators ψ_i:

$$\psi_i(\{*n) = £i,$$
$$\psi_i(+a\{*n) = £i; -a; \#\#£n,$$
$$\psi_i(-a\{*n) = £i; +a; \#\#£n,$$
$$\psi_i(*\}n) = \#\#£n; £i \quad \texttt{with} \quad n > 0.$$

Example:

$$\texttt{pglecwa2pgle}(a; +b\{*5; c; d; *\}2; e) = a; £2; -b; \#\#£5; c; d; \#\#£2; £5; e.$$

9. Structured programming in PGLS

PGLS is the subset of PGLEcw obtained by leaving out termination, labels and goto's. A PGLS program is syntactically correct if its projection to PGLEcwa introduces only

positive annotations. PGLS programs are also called while-programs. PGLS qualifies as a structured program notation by leaving out any form of (unstructured) jumps.

9.1. Comparing PGLEc and PGLS

PGLS is strictly weaker than PGLEc. To see this consider the following PGLEc program that defeats embedding into PGLS:

$$X = £0; +a\{; -e; !; c; \}\{; +e; !; d; \}; \#\#£0.$$

The behavior of X is as follows: $|X| = P_0$, with $P_0 = P_1 \trianglelefteq a \trianglerighteq P_2$, $P_1 = P_3 \trianglelefteq e \trianglerighteq S$, $P_2 = S \trianglelefteq e \trianglerighteq P_4$, $P_3 = c \circ P_0$, $P_4 = d \circ P_0$.

Let K be the collection of PGLS programs Q satisfying the property that either its behavior or the behavior of $Q;!$ is equal to one of the following behaviors P_0, P_1, P_2, P_3 or P_4. These programs perform a 'tail' of the computation of X, involving at least one atomic action. Assume the existence of a PGLS program X_s satisfying $|X| = |X_s|_{\mathrm{pgls}}$. Then $K \neq \emptyset$ and a program (say Y) of minimal length in K must exist. Y cannot be u for a single instruction as none of the behaviors in P_{0-4} is the behavior of a single instruction.

If $Y = Y_1; Y_2$ and $Y_2 \in K$, the program Y is not minimal in its class, as it can be replaced by $Y2$. If $Y = Y_1; Y_2$ (and $Y_2 \notin K$), then $Y_2 =!$ and $Y_1 \in K$, thus contradicting the minimality of Y. To see this notice that programs in PGLS can always terminate by choosing the test results correctly. Therefore $|Y_2|$ is a tail (subproces) of a the behavior $|X|$. As it is not in K it must equal $|!|$.

If $Y = +\alpha\{; Y_1; \}\{; Y_2; \}$, then $\alpha \in \{a, c, d, e\}$ and and either $Y_1 \in K$ or $Y_2 \in K$ (again contradicting minimality). Similarly a negative test cannot serve as the first instruction of Y.

If $Y = \{*; Y_1; *\}$, then Y cannot terminate at all, contradicting the fact that all behaviors in K have terminating computation paths.

Finally if $Y = +\alpha\{*; Y_1; *\}$, then $\alpha = a$. Now a contradiction arises because $|X|$ can terminate after performing a with outcome `true` as well as `false`, whereas Y terminates only after the reply `false`. In other words programs of this form are outside K. A similar problem arises if a negative while loop header is used.

The presence of explicit termination is irrelevant for the proof, a similar example is: $X' = £0; +a\{; -e; \#\#£1; c; \}\{; +e; \#\#£1; d; \}; \#\#£0; £1$.

10. Conclusions

We have proposed PGA, an algebra in which an extremely simple programming language is captured together with an axiomatic semantic model in terms of behaviors. PGA is parameterized by a set Σ of basic instructions.

Both PGA and the semantic equations for the features of PGA are very simple indeed and can easily be memorized. Particular programming languages can be developed either by instantiating the parameter set of PGA or by translating new syntax into PGA (with instantiated parameter).

The virtue of PGA is that it can be used to answer the question 'what is a programming language' by providing a simple and general construction. It can be used to program Turing machines as well as to model simple assembly languages. The authors consider PGA to be a meaningful point of departure for the teaching of programming and software engineering.

The existence of projections does not imply that the 'higher' languages are more expressive then the 'lower' ones. In Appendix A mappings are described in the opposite direction of the projection functions. Such mappings are called embeddings. Without proof it is understood that embeddings preserve semantics, thereby establishing that the embedded notation cannot be strictly more expressive than the target notation of the embedding. In Appendix B an experiment is made with a different form of projection semantics called lazy projection semantics. Lazy projection exploits the structure of PGA. We found that when language complexity increases lazy projection becomes less manageable than (eager) projection, however. Nevertheless lazy projection allows a very simple way to define language extensions for PGA, which justifies its inclusion as an appendix. In Appendix C we investigate the readability and writability of very simple programs. It is suggested that the use of instruction counter comments on top of PGLD allows readability, and that a further extension with templates is needed for writability. In fact PGLDg extends both of these mechanisms, indicating that PGLDg might be useful in 'practical cases', in principle.

Appendix A. Program algebra embeddings

The program algebra projections translate programs in the direction of PGA. In that direction program notations become less and less flexible. There is another line of operators which translate programs in the opposite direction. Such operations will be called program algebra embeddings because these explain the meaning of programs in terms of more flexible program notations. This paper gives rise to six such embeddings: the embedding of PGA expressions into PGLA, the embedding of PGLA into PGLB, the embedding of PGLB into PGLC and the embedding of PGLC into PGLD, the embedding of PGLD into PGLDg and finally the embedding of PGLDg into PGLE.

Unfortunately the embedding `pgla2pglb` is slightly involved due to the case that the repeat instruction may have a parameter larger than the number of instructions preceding it. For that reason a direct embedding `pga2pglb` has been defined and the embedding `pgla2pglb` is based on that by composing it with the program algebra projection for PGA expressions.

A.1. Embedding PGA into PGLA

The program algebra embedding `pga2pgla` is obtained as follows. Given a PGA expression X, one may first bring it into a second canonical form. (This form is not unique, one may settle for the unique form with a shortest repeating and non-repeating parts, however, at the cost of significant computational costs.) Then there are two cases: Y and $Y; Z^\omega$, in which Y and Z do not allow repetition. In the first case we define

$$\texttt{pga2pgla}(Y) = Y.$$

In the second case, $Y; Z^\omega = u_1; \ldots; u_k; (u_{k+1}; \ldots; u_{k+n})^\omega$ and we define

$$\begin{aligned}&\texttt{pga2pgla}(u_1; \ldots; u_k; (u_{k+1}; \ldots; u_{k+n})^\omega)\\ &\quad = u_1; \ldots; u_k; u_{k+1}; \ldots; u_{k+n}; \backslash\backslash\#n.\end{aligned}$$

A.2. Embedding PGA into PGLB

The program algebra embedding pga2pglb is obtained as follows. Again given a PGA expression X, one may first bring it into a second canonical form. Then there are two cases: Y and $Y; Z^\omega$, in which Y and Z do not allow repetition. In the first case we define

$$\texttt{pga2pglb}(Y) = Y.$$

In the second case, $Y; Z^\omega = u_1; \ldots; u_k; (u_{k+1}; \ldots; u_{k+n})^\omega$ and we define

$$\begin{aligned}&\texttt{pga2pglb}(u_1; \ldots; u_k; (u_{k+1}; \ldots; u_{k+n})^\omega)\\ &\quad = u_1; \ldots; u_k; \vartheta_1(u_{k+1}); \ldots; \vartheta_n(u_{k+n}); \backslash\#n; \backslash\#n.\end{aligned}$$

The auxiliary operators ϑ_j are given by

$$\begin{aligned}&\vartheta_j(\#l) = \backslash\#n - l \ \texttt{if} \ \ j + l > n,\\ &\vartheta_j(u) = u \ \texttt{otherwise}.\end{aligned}$$

A.3. Embedding PGLA into PGLB

This embedding is obtained as the composition of the projection pgla2pga and the embedding pga2pglb

$$\texttt{pgla2pglb}(X) = \texttt{pga2pglb}(\texttt{pgla2pga}(X)).$$

A.4. Embedding PGLB into PGLC

The program algebra embedding pglb2pglc is obtained as follows:

$$\texttt{pglb2pglc}(u_1; \ldots; u_k) = \vartheta_1(u_1); \ldots; \vartheta_k(u_k); \#0; \#0.$$

In this case the auxiliary operators ϑ_j are given by

$$\begin{aligned}&\vartheta_j(\#l) = \#0 \ \texttt{if} \ \ j + l > k,\\ &\vartheta_j(\backslash\#l) = \#0 \ \texttt{if} \ \ l \geqslant j,\\ &\vartheta_j(!) = \#k + 3 - j,\\ &\vartheta_j(u) = u \ \texttt{otherwise}.\end{aligned}$$

A.5. Embedding PGLC into PGLD

The program algebra embedding pglc2pgld is defined by

$$\texttt{pglc2pgld}(u_1; \ldots; u_k) = \vartheta_1(u_1); \ldots; \vartheta_k(u_k).$$

In this third case the auxiliary operators ϑ_j are given by

$$\begin{aligned}&\vartheta_j(\#l) = \#\#j + l,\\ &\vartheta_j(\backslash\#l) = \#\#0 \ \texttt{if} \ \ l \geqslant j,\\ &\vartheta_j(\backslash\#l) = \#\#j - l \ \ \texttt{if} \ \ l < j,\\ &\vartheta_j(u) = u \ \texttt{otherwise}.\end{aligned}$$

A.6. Embedding PGLD into PGLDg

The embedding `pgld2pgldg` of PGLD into PGLDg works as follows: let $X = u_1; \ldots; u_k$, then the embedded version is defined as

$$\texttt{pgld2pgldg}(u_1; \ldots; u_k) = £0; \vartheta_1(u_1); \ldots; \vartheta_k(u_k); £k + 1.$$

Here $\vartheta_j(u)$ reads as follows:

$$\begin{aligned}
&\vartheta_j(\#\#l) = £j; \#\#£k + 1 \ \texttt{if}\ l \geqslant k + 1 \ \texttt{or}\ l < 1,\\
&\vartheta_j(\#\#l) = £j; \#\#£l \ \texttt{if}\ l < k + 1,\\
&\vartheta_j(+a) = £j; +a; \#\#£j + 1; \#\#£j + 2,\\
&\vartheta_j(-a) = £j; -a; \#\#£j + 1; \#\#£j + 2,\\
&\vartheta_j(u) = £j; u \ \texttt{otherwise}.
\end{aligned}$$

A.7. Embedding PGLDg into PGLE

Finally PGLDg programs can be embedded in PGLE, thus demonstrating that in spite of the seemingly severe restriction imposed on PGLE, the expressive power of PGLDg and PGLE are identical. An embedding `pgldg2pgle` is as follows: let $X = u_1; \ldots; u_k$. Moreover assume that m is the largest label occurring in a goto instruction or in a label catch instruction in X. Then we read

$$\texttt{pgldg2pgle}(X) = \vartheta_1(u_1); \ldots; \vartheta_k(u_k)$$

with the auxiliary operators ψ_i determined by the following rules:

$$\begin{aligned}
&\vartheta_i(+a) = -a; \#\#£m + i + 1; £m + i,\\
&\vartheta_i(-a) = +a; \#\#£m + i + 1; £m + i,\\
&\vartheta_i(u) = u; £m + i \ \texttt{otherwise}.
\end{aligned}$$

Appendix B. Lazy projection semantics

Projection semantics requires that a program X in a program notation say PGLZ, is transformed entirely into a PGA expression by means of a projection function. A more informative phrase for projection semantics is global projection semantics (another appropriate name is full projection semantics). The projection is globally applied on the entire program. Global projections are like compilers. Lazy projection (also called partial projection) refers to a setting in which instructions outside the instruction set of PGA are transformed into a PGA instruction sequence only when occurring at the first position of a program object. Lazy projection works like an interpreter. Lazy projection allows a program object to be constructed 'on the fly', only transforming non-PGA instructions when needed.

B.1. The unit instruction operator

In this appendix we will briefly describe an operator which is immediately suggested by the setting of PGA, but which is not so standard. The unit instruction operator $u(-)$

takes a PGA program and wraps it into a unit which is taken to have length 1. The length matters, of course, in connection with the evaluation of the effect of jumps and tests. There are no useful program object equations for $u(-)$ known to us. We denote with PGA_u the program notation that extends PGA with the unit instruction mechanism. As it turns out the behavioral semantics of PGA_u can easily be given by adding a single behavior extraction equation to the ones given for PGA

$$|u(X); Y| = |X; Y|.$$

This equation explains how to translate a unit instruction at the head position of a program object.

The description of a full projection for PGA_u is amazingly involved and quite outside the scope of this paper.

B.2. Lazy projections for a conditional instruction and a repetition instruction

PGA can be extended with instructions for a conditional statement and a while loop, thus obtaining PGAcw. These instructions will be given a lazy projection semantics here. Four instructions are used. These instructions are less flexible than the advanced control instructions introduced for PGLEc and PGLEcw. The bonus for the restriction lies in the simplicity of the lazy projection semantics, compared with the full projection semantics given for PGLEc and PGLEcw.

Positive conditional for PGA. $\mathtt{if} + a$; its execution starts with a, if `true` is returned the next action is performed and the subsequent action is skipped; if `false` is returned the next action is skipped and execution continues thereafter.

Negative conditional for PGA. $\mathtt{if} - a$; its execution starts with a, if `false` is returned the next action is performed and the subsequent action is skipped; if `true` is returned the next action is skipped and execution continues thereafter.

Positive while header for PGA. $\mathtt{while} + a$; its execution starts with a. If `true` is returned the next instruction is performed and the whole program is restarted; if `false` is returned the next instruction is skipped and execution continues thereafter.

Negative while header for PGA. $\mathtt{while} - a$; its execution starts with a. If `false` is returned the next instruction is performed and the whole program is restarted; if `true` is returned the next instruction is skipped and execution continues thereafter.

Each of these instructions has its own behavior extraction equation, adequate for providing a lazy projection semantics.

$$|\mathtt{if} + a; u; v; X| = |-a; \#3; u; \#2; v; X|,$$
$$|\mathtt{if} - a; u; v; X| = |+a; \#3; u; \#2; v; X|,$$
$$|\mathtt{while} + a; u; X| = |-a; \#4; u; \mathtt{while} + a; u; X|,$$
$$|\mathtt{while} - a; u; X| = |+a; \#4; u; \mathtt{while} - a; u; X|.$$

These instructions can be used in combination with the unit instruction. For instance: $\mathtt{while} + a; u(\mathtt{if} - b; u(+c; !; d); u(-d; \#0; !); c); b; !$.

B.3. Macros, procedures and recursion

It is an obvious idea to use abbreviations for the description of programs. For $n \in NN$, $\mathtt{p}\,n$ will be used as a program name. A macro definition environment E_k^m of size k provides

for each program name (for $n < k$) a defining equation $\mathtt{p}\,n = P_n$ with P_n a PGLA program (or a program in a suitable extension for which at least a lazy projection is known). A macro instruction has the form $m(\mathtt{p}\,n)$ with $n < k$. PGA may be extended with macro instructions (thus obtaining PGLAm). Given some macro definition environment, a lazy projection for macro instructions is given by

$$|m(\mathtt{p}\,n);\, X| = |P_n;\, X|.$$

A full projection is obtained by transforming each macro instruction $m(\mathtt{p}\,n)$ into $u(P_n)$.

Macros disallow the occurrence of other macros in the defining programs P_n. It is an obvious possibility to relax that constraint. A procedure call instruction has the form $c(\mathtt{p}\,n)$. A procedure definition environment of size $k(E_k^p)$ contains for each program name $\mathtt{p}\,n$ (with $n < k$) a defining identity $\mathtt{p}\,n = P_n$ where the programs P_n may contain procedure call instructions as well (thus allowing recursion). The same behavior extraction equation that applies for macros applies in this case: given some procedure definition environment, a lazy projection for macro instructions is given by

$$|c(\mathtt{p}\,n);\, X| = |P_n;\, X|.$$

Though remarkably similar to the case of macros, the lazy projection for procedures cannot be replaced by a full projection.

Appendix C. Comment instructions

The rationale of the program algebra language family as outlined above is primarily to provide program notations that are as simple as possible, to provide a very clear and concise behavioral semantics and to have a platform for the understanding of assembly-level programming and code generation. No attention has been paid to optimizing or even evaluating these formats as carriers for programs in a form that can be read, understood or even written by human agents. The languages PGLA-E have several drawbacks from a human point of view.

C.1. Human readers and writers

As a consequence it is not entirely clear where in the hierarchy of languages built on top of PGLA, the first or simplest notation is to be found that can be used by a programmer. This is a pragmatic question. We have concluded that programming directly in PGLA (or PGA) is virtually impossible because the bookkeeping of the jumps will soon be too complex. The same holds for PGLB and for PGLC, albeit to a slightly lesser extent. In both cases it is too hard to get the jump counters right. In particular it is unclear how to make a stepwise development in which jump counters can be adapted. A solution may be found in changing the language PGLD to PGLDg by adding the feature of goto's (and removing the then obsolete absolute jump instructions). PGLDg, thus obtained, is suitable for human usage in simple circumstances. However, goto's and labels are not entirely straightforward because of alpha-conversion for labels, which introduces a non-trivial level of abstraction.

C.2. Comment instructions for PGLD

We will propose comment instructions. The language PGLD extended with the comment instruction option is called PGLDco. These comments will help a human reader of a

PGLDco program. The syntax of comment instructions is as follows: % is an empty comment instruction and for a string s that does not contain ';' % s is the comment instruction that gives comment 's'.

The relevance of comment instructions is to provide a human reader with information that is immaterial for the operational meaning of the program. For people to understand the meaning of the program, a projection function is needed which translates programs from the extended language PGLDco back to PGLD. This is the function `pgldco2pgld`, which works as follows:

$$\texttt{pgldco2pgld}(u_1; \cdots; u_k) = \psi_1(u_1); \cdots; \psi_k(u_k).$$

The auxiliary operations ψ_i read as follows:

$$\psi_i(\%) = \#\#i + 1,$$
$$\psi_i(\%\ s) = \#\#i + 1,$$
$$\psi_i(u) = u \quad \texttt{otherwise}.$$

If instruction number i equals $\#\#i + 1$, its effect upon execution is to pass control to the next instruction without any further side effect. The behavior of a PGLDco program X can be defined as $|X|_{\text{pgldco}} = |\texttt{pgldc2pgla}(X)|_{\text{pgld}}$.

Comments are a tool for readers, which does not necessarily make them a tool for authors as well. In the case of PGLDco we will outline a kind of comment, the instruction count comment, which helps the reader much more than the author. In fact, the author is still faced with the original problem of having to compute the right counter of a jump while programming. The comment mentioned is as follows: '% `ic` k' is an instruction which asserts 'this is instruction number k'. A PGLDco program is instruction count comment correct (ICCC) if all instruction count comments assert the right thing about their position in the program. So $a; b;$ %`ic` $3; +b;$ ##3 is ICCC, whereas $a; b; c;$ %`ic` $3; +b;$ ##3 is not ICCC. An author who presents a program to a reader will try to ensure that it is ICCC. If not, the reader will feel free not to examine the program in further detail. The test is easy, but boring. The advantage of ICCC PGLDco programs is simply that a reader need not calculate what the jumps point to. In fact, the jump counters are like labels, albeit not in a mnemonic style. Although one can do without, the ease provided by mnemonic names is considerable.

The following example serves to illustrate the burden which is put on the reader by the use of jump counters, without instruction counter comments

$$a; b; -g; \#\#21; b; b; \#\#14; +d; -e; +g; !; !; !; \%\texttt{ic}\ 14; a; a; a; b; b; b; \%\texttt{ic}\ 21; f$$

which has the same behavior as

$$a; b; -g; \#\#20; b; b; \#\#14; +d; -e; +g; !; !; !; a; a; a; b; b; b; f.$$

We consider the overhead in program length to be justified by the additional information concerning the targets of the jumps.

C.2.1. PGLDcot, a design language for PGLDco

An author who is satisfied with PGLDco programs as textual products to be presented to his/her audience still faces the problem of having to write these programs. Now it is quite likely that the author knows how to proceed if the use of labels were allowed. The difficulty is that at the time of writing down an absolute jump instruction (in particular

one that actually is a forward jump) it may not exactly be known to which instruction the jump has to move control. The standard solution to such matters is to take advantage of the philosophy of top-down design, and to use a template in preparation of the construction of the program. Templates stand for the real thing, which will be substituted when more information is available. In the case of PGLDco two template instructions are plausible: '##x' and '%ic x' where x is a variable for natural numbers. By allowing these extra instructions the program design notation PGLDcot is obtained.

A design that can precede the example program above then is

$$a; b; -g; \#\#x1; b; b; \#\#x2; +d; -e; +g; !; !; !; \%\texttt{ic}\ x2; a; a; a; b; b; b; \%\texttt{ic}\ x1; f.$$

After having made this design, the author of the program will subsequently compute $x1 = 21$ and $x2 = 14$, and later substitute these values. In fact the design may be even easier to read than its translation into PGLDco, but there is a simple reason not to use these designs in some occasions. The cost of explaining the additional complexity of the variable binding may be prohibitive. As it is in general to be expected that for every program notation, the best design notation for it is a proper extension, one is not tempted to take a design notation as a means of presentation. Admittedly the cost of explaining a particular design in PGLDcot is not very high: each variable must be instantiated with the (unique) instruction number of an instruction that features the variable in an instruction count comment. This can be formalized in the definition of a transformation `pgldcot2pgldc`. The requirements of ICCC are more severe for PGLDcot expressions. Not only should the instantiated instruction counts be correct, the template variables should all have a unique defining occurrence in a comment instruction. The test on ICCC and the transformation of a design expression to a PGLDco program can well be supported by an automated programming environment. The design language itself can also be viewed as a program notation, but the cost of its introduction may be considered prohibitive in some cases.

C.2.2. Using the design notation

Having the design notation available, the programmer may decide to first write a design and subsequently transform it into a program. In the case of PGLDco this strategy may well work for programs of 100 instructions or less. For much longer programs the readability of PGLDco is doubtful and the transformation from a design expression (written in PGLDcot) which makes use of well-chosen mnemonic jump counter names, to an expression that uses instantiated jump counters may be unreasonable. The problems appearing when having to present programs of increasing size explain many of the features that have been developed for modern programming notations. In principle there is an unbounded supply of possibilities to find short presentations of (previously long) programs.

What is clear from the above considerations is that the aspect of human readability, the use of comments and the distinctions of languages for engineering (design), presentation (with comments) and execution (without comments) is equally relevant for extremely simple languages. Some programming languages seem to be more complicated than necessary because an attempt has been made to include the features needed for design in the language itself. It can be expected that the more complex a language becomes, the more discrepancies between syntactically correct programs and intermediate design expressions will appear.

C.3. Other comments

Of course, more comments are conceivable. For instance, in PGLDco it may be useful to have comments that simply provide a name for a section in a program or the date of production, the name of the author or any other useful kind of information. The name of a program can also be expressed in a comment. A systematic naming of such comments can be developed, but it will not be a concern in program algebra.

In other program notations based on program algebra, comments can be introduced in a similar way, provided that the projection removing the comments makes use of an instruction that has no side-effect and passes control to the next one. This instruction depends on the program notation used. For instance in PGLA, PGLB and PGLC it is the forward jump with counter 1: #1, and in a language with goto's it may be the catch of a label that is not used.

C.3.1. Concluding remarks

The decision to see comments as instructions is needed if programs are to be instruction lists and commented programs are to be viewed as programs as well. This is plausible because an explosion of the number of conceptual categories is not useful. For the same reason it is plausible to treat the design notation PGLDcot for PGLDco simply as another program notation and to regard carrying out the transformation `pgldcot2pgldc` as a task for the programmer (in some cases). As long as such a transformation can be easily automated there is no need to view design notations as a conceptually different category. A different situation arises if the transformation from design to program is not computable or clearly requires human intervention because of experience-based steps that one has not been able to capture in a more formal way.

Appendix D. FMN: Focus method notation

The purpose of this appendix is to provide a fixed notational format for basic instructions sufficiently expressive for all applications we have in mind.

D.1. Instruction phases

The question 'what is an instruction?' refutes any simple answer. In order to see the nature of the problem one may consider the question: what is water? will ice qualify or must it be a fluid? As a matter of fact water can exist in different phases allowing transitions from one phase to another. For basic instructions one may imagine different phases as well. In that case the following phases may be distinguished (while many more phases can be imagined): (type) written phase, binary phase, stored binary phase, fetched binary phase, execution phase. FMN addresses the typewritten phase only, focussing on representing instructions (and programs) as logical sequences of ASCII characters.

D.2. Focus and method

FMN basic instructions may either have a focus or not. If no focus is present an execution architecture will use a default focus instead. A focus represents a part of a system able to process a basic instruction and to respond subsequently with a boolean value. Such a

part may e.g. be called a reactor, a coprogram or an instruction execution agent. The second part of an instruction with focus (and the only part of an instruction without focus) consists of a method. Focus and method are combined by means of a '.'. Focus and method may both consist of alphanumeric ASCII sequences, starting with a letter from the alphabet and allowing a colon (`:`) as a separator of parts. Here are some possible typewritten instructions:

```
a,
print:file:c3,
printer:32.print:file:a2,
registers:3.assign:x:to:y,
stack:3.Push:5,
Stack:2.pop,
table:2.insert:5:at:2
ab::.c::
```

A formal CF grammar of FMN is omitted. If basic instructions are taken from FMN and programs are given in PGLA, the resulting notation is termed PGLA:FMN (or pgla:fmn). Here is a PGLB:FNM program:

```
+a2;Bb.de:true;\#1;-a:2.b:3;#5;A:true.false:5.
```

Acknowledgements

Jan Bergstra acknowledges a significant suggestion by Anne Kaldewaij, and substantial suggestions by Alban Ponse, Piet Rodenburg and Ruud Roël.

References

[1] J.A. Bergstra, J.-W. Klop, Process algebra for synchronous communication, Inf. Control 60 (1/3) (1984) 109–137.

[2] J.A. Bergstra, M.E. Loots, Program algebra for component code, Formal Aspects Comput 12 (1) (2000) 1–17.

[3] W.J. Fokkink, Axiomatizations for the perpetual loop in process algebra, in: P. Degano, R. Gorrieri, A. Marchetti-Spaccamela (Eds.), Proceedings of the 24th Colloquium on Automata, Languages and Programming-ICALP'97, Lecture Notes in Computer Science, vol. 1256, Springer, Berlin, 1997, pp. 571–581.

[4] C.A.R. Hoare, I.J. Hayes, He Jifeng, C.C. Morgan, A.W. Roscoe, J.W. Sanders, I.H. Sorensen, J.M. Spivey, B.A. Sufrin, Laws of programming, Commun. ACM 30 (8) (1987) 672–686.