ELSEVIER

The Journal of Logic and Algebraic Programming 51 (2002) 157–174

THE JOURNAL OF LOGIC AND ALGEBRAIC PROGRAMMING

www.elsevier.com/locate/jlap

Program algebra with unit instruction operators

Alban Ponse*

Programming Research Group, Faculty of Science, University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
CWI, Department of Software Engineering, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

Abstract

In the setting of program algebra (PGA), a projection from $\mathrm{PGA}_{\mathrm{u}}$, i.e., PGA extended with a unit instruction operator, into PGA is defined. This is done via a composition that employs backward jumps and (labeled) goto's. © 2002 Elsevier Science Inc. All rights reserved.

Keywords: Program algebra; Unit instruction operator; Projection semantics

1. Introduction

Program algebra (PGA) is an area of research that provides an algebraic framework and semantical foundations for sequential programming in assembly-like programming languages. In [1], PGA is defined as a basic notation for such languages. Furthermore, that paper introduces a family of languages comprising more advanced programming features. These languages are systematically interrelated via projections (from 'higher' dialects into PGA) and embeddings (mappings in the reverse direction). Motivation for PGA and further information can be found in [1,2].

In [1,2] it is observed that the *unit instruction operator*, which takes a PGA program and wraps it into a unit of length one, is a natural extension of PGA. This length is significant for the evaluation of jumps and tests. In this paper a projection from PGA extended with the unit instruction operator into PGA is defined. The existence of such a projection implies that the unit instruction operator is not needed as a primitive in terms of expressiveness. Nevertheless, this operation is of interest because:

(1) It allows for a much more flexible style of programming (just as the PGA-based program notations with more advanced jump instructions that are closer to programming practice).
(2) It may be a useful tool in the study of program algebra itself.

* Tel.: +31-20-525-7592; fax: +31-20-525-7490.
E-mail address: alban@science.uva.nl

PII: S1567-8326(02)00019-X

The first property is illustrated by two running examples throughout the paper, while the second is demonstrated in the last two sections, where *composed instructions* are defined, and analyzed with help of the unit instruction operator.

The structure of this paper is as follows: in Section 2 some basic facts about PGA and program equivalence are recalled. In Section 3 the behavioral semantics for PGA programs as defined in [2] is summarized. In Section 4 some variants of PGA that are used to define the above-mentioned projection are introduced, and the projection itself is defined in Section 5. This paper is ended with some conclusions and discussion in Section 6.

2. The language PGA and instruction sequence congruence

In this section some basic information on PGA (taken from [2]) is recalled.

2.1. Basics of PGA

The programming language PGA is based on a parameter set Σ of the so-called *basic instructions*. These are regarded as indivisible units and execute in finite time. Furthermore, a basic instruction is viewed as a request to the environment, and it is assumed that upon its execution a boolean value (`true` or `false`) is returned that may be used for subsequent program control. The language PGA has two composition constructs:

Concatenation. If X and Y are programs (or 'program terms'), i.e., closed terms, then $X; Y$ is one as well.

Repetition. If X is a program, so is $(X)^\omega$. If no confusion can arise, the brackets in a repetition may be dropped, e.g. if $X = \mathtt{a}$, where $\mathtt{a}$ is a basic instruction, then X^ω stands for $\mathtt{a}^\omega$ and if $X = \mathtt{a}; \mathtt{a}$, then X^ω stands for $(\mathtt{a}; \mathtt{a})^\omega$.

Given Σ, the primitive instructions of PGA are the following:

Void basic instruction. All elements of Σ, typically $\mathtt{a}, \mathtt{b}, \ldots$ are such instructions. When executed, a void basic instruction generates a boolean value and the associated behavior may modify a state. After execution, a program has to enact its subsequent instruction. If that instruction fails to exist, inaction occurs. The attribute void expresses that subsequent execution is not influenced by the returned boolean value.

Termination instruction. The termination instruction ! yields termination of the program. It does not modify a state, and it does not return a boolean value.

Positive test instruction. For each element $\mathtt{a}$ of Σ there is a positive test instruction $+\mathtt{a}$. When executed, the state is affected according to $\mathtt{a}$, and in case `true` is returned, the remaining sequence of actions is performed. If there are no remaining instructions, inaction occurs. In the case that `false` is returned, the next instruction is skipped and execution proceeds with the instruction following the skipped one. If no such instruction exists, inaction occurs.

Negative test instruction. For each element $\mathtt{a}$ of Σ there is a negative test instruction $-\mathtt{a}$. When executed, the state is affected according to $\mathtt{a}$, and in case `false` is returned, the remaining sequence of actions is performed. If there are no remaining instructions, inaction occurs. In the case that `true` is returned, the next instruction is skipped and execution proceeds with the instruction following the skipped one. If no such instruction exists, inaction occurs.

Forward jump instruction. For any natural number k, the instruction $\#k$ denotes a jump of length k and k is called the counter of this instruction. If $k = 0$, this jump is to the instruction itself and inaction occurs (one can say that $\#0$ *defines* divergence, which is a particular form of inaction). If $k = 1$, the instruction skips itself, and execution proceeds with the subsequent instruction if available, otherwise inaction occurs. If $k > 1$, the instruction $\#k$ skips itself and the subsequent $k - 1$ instructions. If there are not that many instructions left in the remaining part of the program, inaction occurs.

Note that with *unfolding*, captured by the identity $X^\omega = X; X^\omega$ and explained in Section 2.2, PGA programs refer to an execution mechanism that is left-sequential (from left to right). This is closer to the behavioral semantics defined in [2] (and discussed in Section 3) than would be possible when more 'advanced' programming features as *goto*'s or *backward jumps* were included from the start, and hence may clarify why PGA is distinguished as most basic.

2.2. Instruction sequence congruence and canonical forms

On PGA, different types of equality can be discerned, the most simple of which is *instruction sequence congruence*, identifying programs that execute identical sequences of instructions. Such a sequence is further called a *program object*. For programs not containing repetition, instruction sequence congruence boils down to the associativity of concatenation, and is axiomatized by

$$\text{(PGA1)} \qquad (X; Y); Z = X; (Y; Z).$$

As a consequence, brackets are not meaningful in repeated concatenations and will be left out. Now let $X^1 = X$ and for $n > 0$, $X^{n+1} = X; X^n$. Then instruction sequence congruence for infinite program objects is further axiomatized by the following axioms (schemes):

$$\text{(PGA2)} \qquad (X^n)^\omega = X^\omega,$$
$$\text{(PGA3)} \qquad X^\omega; Y = X^\omega,$$
$$\text{(PGA4)} \qquad (X; Y)^\omega = X; (Y; X)^\omega.$$

It is straightforward to derive from (PGA2) to (PGA4) the unfolding identity of repetition: $X^\omega = (X; X)^\omega = X; (X; X)^\omega = X; X^\omega$. Whenever two programs X and Y are instruction sequence congruent, this is written $X =_{\text{isc}} Y$. The subscript $_{\text{isc}}$ will be dropped if no confusion can arise. Instruction sequence congruence is decidable (see [2]).

Each PGA program term can be rewritten into one of the following forms:
(1) Y not containing repetition, or
(2) $Y; Z^\omega$, with Y and Z not containing repetition.

Any program term in one of the two above forms is said to be in *first canonical form*. According to [1,2], for each closed PGA term there is a PGA program term in first canonical form that is instruction sequence congruent. Canonical forms are useful as input for further transformations.

For concise representation, Y^ω with Y a program term not containing repetition is also considered a PGA program term in first canonical form in the remainder of this paper. Note that $Y^\omega = Y; Y^\omega$, and the right-hand side equals form (2) above.

3. Behavioral semantics for PGA programs

In this section the behavioral semantics defined in [2] is summarized. This semantics is based on BPPA, basic polarized process algebra.

3.1. Primitives of BPPA

As is the case with PGA and its programming language PGA, BPPA is based on a collection Σ of basic instructions, called 'actions' in the setting of behavioral semantics. BPPA has two constants and two composition mechanisms, and is equipped with a family of approximation operators. The constants model termination and inaction. Given Σ, BPPA_Σ denotes its associated family of program behaviors.

Termination. With $\mathtt{S}$ (stop) the terminating behavior is denoted; it does no more than terminate, and has no side effect on a state.

Divergent behavior. By $\mathtt{D}$ (inaction or divergence) an inactive behavior is indicated. It is a behavior that represents the impossibility of making real progress (an example of this is a loop resulting from an infinite number of consecutive jumps, as in $\#\mathtt{0}$ or $(\#\mathtt{1})^\omega$, not yielding any observable 'activity'). Like termination, inaction does not affect a state in which it occurs.

The constants $\mathtt{S}$ and $\mathtt{D}$ are contained in BPPA_Σ. The composition mechanisms are *postconditional composition* and *action prefix*, where action prefix is an abbreviation:

Postconditional composition. For action $\mathtt{a} \in \Sigma$ and behaviors P and Q in BPPA_Σ,

$$P \trianglelefteq \mathtt{a} \trianglerighteq Q$$

denotes the behavior in BPPA_Σ that first performs $\mathtt{a}$ and then either proceeds with P if $\mathtt{true}$ was produced, and otherwise with Q.

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

$$\mathtt{a} \circ P = P \trianglelefteq \mathtt{a} \trianglerighteq P.$$

3.2. Approximation of program behaviors

A program behavior is called finite if there is a finite upper bound to the number of consecutive actions it can perform. Finite behaviors are made from $\mathtt{S}$ and $\mathtt{D}$ by means of postconditional composition. The definition of infinite behaviors makes use of the so-called 'projective sequences'. These in turn require *approximation operators* π_n ($n \in \mathbb{N}$), which are defined as follows:

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

and hence, $\pi_{n+1}(\mathtt{a} \circ P) = \mathtt{a} \circ \pi_n(P)$.

A *projective sequence* is a sequence $(P_n)_{n\in\mathbb{N}}$ such that for each $n \in \mathbb{N}$,

$$\pi_n(P_{n+1}) = P_n.$$

Projective sequences can be used to represent finite as well as infinite behaviors, and are considered equal exactly if all components are equal. A finite behavior P is represented

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

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 n-th approximations of the two behaviors are equal. Finite approximations of behaviors are considered equal if and only if they have exactly the same form.

3.3. Behavior extraction and behavioral equivalence

Semantic equations define the behavior of complex programs in terms of the behavior of their constituent parts. The *behavior extraction operator* $|\_|$ assigns a behavior to a program. Instruction sequence congruent programs have identical behaviors, but the behavioral equivalence defined by behavior extraction is not a congruence, i.e., from the fact that $|X|$ and $|Y|$ are the same behavior, one cannot infer that $|X; Z|$ and $|Y; Z|$ are the same behavior (or $|Z; X|$ and $|Z; Y|$, or $|X^\omega|$ and $|Y^\omega|$).

For any finite program object X, its behavior is determined by

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

expressing that if the program ends without being able to perform an explicit termination instruction, the program execution stagnates (which is modelled as inaction). With this identity and unfolding, each PGA program behavior matches exactly one of the semantic equations below. In these equations, $\mathtt{a}$ ranges over the basic instructions in Σ, u ranges over all primitive instructions and X ranges over arbitrary program objects:

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

The semantic equations for jump instructions require a case distinction on the counter of the jump. In case the counter is zero, inaction will occur. In case the counter is one, at least one further instruction should be present, otherwise inaction occurs. In case the counter exceeds one, the program should contain at least two subsequent instructions; otherwise the program becomes inactive. In the equations below, k ranges over the natural numbers.

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

The above equations should be used to obtain successive steps of the behavior of a program object X. These equations may never generate atomic behavior. In that case the program has a non-trivial loop and its behavior will be identified with $\mathtt{D}$, for instance: $|(\#\mathtt{1})^\omega| = \mathtt{D}$ and $|\mathtt{b}; (\#\mathtt{2}; \mathtt{a})^\omega| = \mathtt{b}\circ\mathtt{D}$. Phrased differently: if for a behavior $|X|$ the behavior extraction equations fail to prove $|X| = \mathtt{S}$ or $\pi_1(|X|) = \mathtt{a}\circ\mathtt{D}$ for some $\mathtt{a} \in \Sigma$, then $|X| = \mathtt{D}$.

If X has no repetition, $|X|$ is a finite behavior. Programs with repetition can have infinite behaviors. As an example consider $\mathtt{a}^\omega$. The equations above yield $|\mathtt{a}^\omega| = \mathtt{a} \circ \mathtt{a} \circ \mathtt{a} \circ \cdots$. Using projective sequence notation: $|\mathtt{a}^\omega| = (P_n)_{n\in\mathbb{N}}$ with $P_0 = \mathtt{D}$, $P_1 = \mathtt{a} \circ \mathtt{D}$, $P_2 = \mathtt{a} \circ \mathtt{a} \circ \mathtt{D}, \cdots$. A concise characterization of $|\mathtt{a}^\omega|$ is captured by the recursive equation

$$|\mathtt{a}^\omega| = \mathtt{a} \circ |\mathtt{a}^\omega|.$$

Two programs X and Y are behaviorally equivalent (denoted by $X \equiv_{\mathrm{be}} Y$) if $|X| = |Y|$. This in turn holds precisely if for all $n \in \mathbb{N}$, $\pi_n(|X|) = \pi_n(|Y|)$. It can be shown that it is decidable whether or not $X \equiv_{\mathrm{be}} Y$ for closed PGA program terms X and Y (see [2]). Behavioral equivalence includes instruction sequence congruence, i.e., if $X =_{\mathrm{isc}} Y$, then $X \equiv_{\mathrm{be}} Y$. As an example,

$$|\mathtt{b}^\omega; \mathtt{c}^\omega| = |\mathtt{b}^\omega| \tag{1}$$

or, equivalently, $\mathtt{b}^\omega; \mathtt{c}^\omega \equiv_{\mathrm{be}} \mathtt{b}^\omega$, because this is an instance of axiom (PGA3). Behavioral equivalence is non-compositional,[1] e.g., $+\mathtt{a} \equiv_{\mathrm{be}} \mathtt{a}$, while $+\mathtt{a}; \mathtt{b} \not\equiv_{\mathrm{be}} \mathtt{a}; \mathtt{b}$. For another example, $\#2; \mathtt{a} \equiv_{\mathrm{be}} \#3; \mathtt{a}$, but $(\#2; \mathtt{a})^\omega \not\equiv_{\mathrm{be}} (\#3; \mathtt{a})^\omega$.

4. The unit instruction operator in PGA and PGLB

In this section the *unit instruction operator*, introduced in [1], is discussed. This operator takes a PGA program and wraps it into a unit of length one. This length matters in connection with the evaluation of jumps and tests. The extension of PGA with the unit instruction operation is denoted by $\mathrm{PGA_u}$. Furthermore, some variants of PGA that will be used to define a projection semantics for $\mathrm{PGA_u}$ are described.

4.1. PGA_u and its canonical forms

The unit instruction operator, notation $\mathbf{u}(\_)$, allows for a flexible style of PGA-programming. As an example,

$$+\mathtt{a}; \mathbf{u}(\mathtt{b}^\omega); \mathtt{c}^\omega \tag{2}$$

has the behavior $|\mathtt{b}^\omega| \trianglelefteq \mathtt{a} \trianglerighteq |\mathtt{c}^\omega|$, as will be explained below. Like repetitions, units are semipermeable, but in a complementary sense: whereas a jump to a non-starting position in a repetition is possible and a jump out of a repetition is not, a jump out of a unit is possible, but a jump to a non-starting position in the unit is not.

Following the intuitions given thus far, the behavior of $\mathrm{PGA_u}$ programs might be defined by the following equation:

$$|\mathbf{u}(X); Y| = |X; Y|,$$

because once execution has entered the body of a unit, the unit has become transparent. The behavioral extraction defined by this equation is called the *lazy projection* of $\mathrm{PGA_u}$ into PGA (cf. [2]). With lazy projection the behavior of the program (2) above can be deduced as follows:

$$|+\mathtt{a}; \mathbf{u}(\mathtt{b}^\omega); \mathtt{c}^\omega| = |\mathbf{u}(\mathtt{b}^\omega); \mathtt{c}^\omega| \trianglelefteq \mathtt{a} \trianglerighteq |\mathtt{c}^\omega|$$

[1] This is the reason to use the notation $\equiv_{\mathrm{be}}$ rather than $=_{\mathrm{be}}$.

$$= |\mathtt{b}^\omega;\, \mathtt{c}^\omega| \trianglelefteq \mathtt{a} \trianglerighteq |\mathtt{c}^\omega|$$
$$\overset{(1)}{=} |\mathtt{b}^\omega| \trianglelefteq \mathtt{a} \trianglerighteq |\mathtt{c}^\omega|. \tag{3}$$

For another example, lazy projection yields that the behavior of the program

$$(+\mathtt{a};\, \mathbf{u}(+\mathtt{b};\, \#5;\, !;\, \mathtt{c});\, \mathtt{d};\, +\mathtt{e})^\omega \tag{4}$$

is captured by the following recursive equation:

$$|X| = (|X| \trianglelefteq \mathtt{b} \trianglerighteq \mathtt{S}) \trianglelefteq \mathtt{a} \trianglerighteq \mathtt{d} \circ (|X| \trianglelefteq \mathtt{e} \trianglerighteq (|X| \trianglelefteq \mathtt{b} \trianglerighteq \mathtt{S})), \tag{5}$$

where action prefix $\circ$ is taken to bind stronger than postconditional composition. In Section 5.4 we arrive at the same characterizations (3) and (5) via a different route, namely by using the full (or total) projection function that is defined in the following section. Lazy projection is further discussed in Section 6.1.

The notion of the first canonical form for PGA programs (see Section 2.2) immediately extends to PGA with units: a PGA_u program is in first canonical form if it is when units are regarded as primitive instructions, and the bodies of all units are in first canonical form as well. The example programs (2) and (4) above both are in first canonical form. (Recall that also b^ω is considered a first canonical form in this paper.)

4.2. PGLB, PGLB_g and PGLB_u

In this paper some variants of PGA are used to define a projection semantics for PGA_u. The most basic of these is the program notation PGLB (see [1,2]), which is defined by adding backward jumps to PGA and omitting repetition (which has then become a redundant feature):

Backward jump instruction. For any natural number k, the instruction $\backslash\#k$ denotes a backwards jump of length k. If $k = 0$, this jump is to the instruction itself and inaction occurs. If $k > 0$, the instruction $\backslash\#k$ moves execution to proceed at k instructions backwards. If there are not that many instructions in the preceding part of the program, inaction occurs.

The program notation PGLB_g (PGLB with labels and goto's) is defined as a variant of PGLB by leaving out the forward and backward jumps, and adding labels and goto's[2]. Assume a decidable and infinite set of labels as a (second) parameter of PGLB_g. The added instructions are these:

Label catch instruction. The label catch instruction has the form $\mathtt{L}\sigma$ for σ some label. Upon execution, this instruction is simply passed and cannot modify a state. If there is no subsequent instruction to be executed, inaction occurs.

Absolute goto instruction. This instruction takes the form $\#\#\mathtt{L}\sigma$ for σ some label, and represents a jump to the leftmost occurrence of the label catch instruction $\mathtt{L}\sigma$ in the program. If there is no such instruction, inaction occurs.

The language PGLB_g can be seen as an extension of PGLB, in which the latter can be embedded: it is not hard to add (forward and backward) jumps to PGLB_g, but this is not done as these can simply be emulated.

Finally, the extension of PGLB with the unit instruction operation is denoted by PGLB_u. The reason to consider PGLB_u (next to PGA_u) is that units have a clear-cut (syntactical) internal length (number of instructions): a length measure on repetitions is not anymore an issue. The internal length of a unit $\mathbf{u}(X)$, i.e. the number of instructions of X (in which

[2] This extension reflects the one defined in [1] on PGLD.

occurrences of $\mathbf{u}(\_)$ are counted as single instructions), is further called its *unit-length*. Having units with a fixed, finite unit-length, one can keep track of the position within a unit, and for forward jumps also of the unit-length of all encompassing units.

5. Projecting $\mathrm{PGA_u}$ into PGA

In this section a (relatively simple) program algebra projection function from $\mathrm{PGA_u}$ into PGA is described. Following the notational conventions in [1], the projection from $\mathrm{PGA_u}$ into PGA is denoted by pgau2pga (from $\mathrm{PGA_u}$ *to* PGA). This projection is defined as a composition of four mappings (embeddings or projections):

$$\begin{array}{ccccc} \mathrm{PGLB_u} & \xrightarrow{\texttt{pglbu2pglbg}} & \mathrm{PGLB_g} & \xrightarrow{\texttt{pglbg2pglb}} & \mathrm{PGLB} \\ \uparrow{\scriptstyle \texttt{pgau2pgbu}} & & & & \downarrow{\scriptstyle \texttt{pglb2pga}} \\ \mathrm{PGA_u} & & \xrightarrow{\texttt{pgau2pga}} & & \mathrm{PGA} \end{array}$$

where the projection pglbu2pglb constitutes the algorithmic kernel. In the following sections each of these mappings is described in detail. (Except for the projection pglb2pga these mappings were not defined before.)

5.1. Embedding $\mathrm{PGA_u}$ in $\mathrm{PGLB_u}$

The embedding pgau2pgbu is defined on program terms in first canonical form (see Section 4.1). For $k, n > 0$,

$$\begin{aligned} &\texttt{pgau2pgbu}(u_1; \ldots; u_k) = \psi(u_1); \ldots; \psi(u_k), \\ &\texttt{pgau2pgbu}((u_1; \ldots; u_n)^\omega) = \psi(u_1); \ldots; \psi(u_n); (\backslash\#n)^{\max(m,2)} \end{aligned}$$

where m is the maximum of the jump counters occurring in $u_1; \ldots; u_n$ and 0 otherwise,

$$\begin{aligned} &\texttt{pgau2pgbu}(u_1; \ldots; u_k; (u_{k+1}; \ldots; u_{k+n})^\omega) \\ &\quad = \psi(u_1); \ldots; \psi(u_k); \psi(u_{k+1}); \ldots; \psi(u_{k+n}); (\backslash\#n)^{\max(m,2)} \end{aligned}$$

where m is the maximum of the jump counters occurring in $u_1; \ldots; u_{k+n}$ and 0 otherwise, and where the auxiliary operation ψ is defined as follows:

$$\begin{aligned} &\psi(\mathbf{u}(X)) = \mathbf{u}(\texttt{pgau2pgbu}(X)), \\ &\psi(u) = u \text{ otherwise.} \end{aligned}$$

Application of pgau2pgbu to the two previously mentioned examples yields:

$$\begin{aligned} &\texttt{pgau2pgbu}(+\texttt{a}; \mathbf{u}(\texttt{b}^\omega); \texttt{c}^\omega) \\ &\quad = +\texttt{a}; \mathbf{u}(\texttt{pgau2pgbu}(\texttt{b}^\omega)); \texttt{c}; \backslash\#1; \backslash\#1 \\ &\quad = +\texttt{a}; \mathbf{u}(\texttt{b}; \backslash\#1; \backslash\#1); \texttt{c}; \backslash\#1; \backslash\#1, \\ &\texttt{pgau2pgbu}((+\texttt{a}; \mathbf{u}(+\texttt{b}; \#5; !; \texttt{c}); \texttt{d}; +\texttt{e})^\omega) \\ &\quad = +\texttt{a}; \mathbf{u}(+\texttt{b}; \#5; !; \texttt{c}); \texttt{d}; +\texttt{e}; \backslash\#4; \backslash\#4; \backslash\#4; \backslash\#4; \backslash\#4. \end{aligned}$$

5.2. Embedding $PGLB_{\mathrm{u}}$ in $PGLB_{\mathrm{g}}$

The embedding $\mathtt{pglbu2pglbg}$ is defined inductively, where sequences of natural numbers are used as labels: the empty sequence is written ϵ, and "," is used as a separator between the natural numbers occurring in a sequence. First,

$$\mathtt{pglbu2pglbg} \triangleq \mathtt{pglbu2pglbg}^{\epsilon}_{\epsilon}.$$

In the definition of $\mathtt{pglbu2pglbg}^{\rho}_{\sigma}$ below, the subscripted sequence σ is used to keep track of the relative position in a unit and that of all encompassing units, while the superscripted sequence ρ records the current unit-length and that of all encompassing units. Note that by definition, σ and ρ have equal length. The embedding $\mathtt{pglbu2pglbg}^{\rho}_{\sigma}$ uses auxiliary functions $\mathtt{f\_target}$ (forward target) and $\mathtt{b\_target}$ (backward target) that compute the label of goto's, and is defined as follows:

$$\mathtt{pglbu2pglbg}^{\rho}_{\sigma}(u_1, \ldots, u_k) = \vartheta^{\rho}_{1,\sigma}(u_1); \ldots; \vartheta^{\rho}_{k,\sigma}(u_k),$$

where the auxiliary operation $\vartheta^{\rho}_{j,\sigma}(u)$ is defined by:

$$\begin{aligned}
&\vartheta^{\rho}_{j,\sigma}(\#l) = \mathtt{L}j, \sigma; \#\#\mathtt{Lf_target}(l, (j, \sigma), \rho),\\
&\vartheta^{\rho}_{j,\sigma}(\backslash\#l) = \mathtt{L}j, \sigma; \#\#\mathtt{Lb_target}(l, (j, \sigma)),\\
&\vartheta^{\rho}_{j,\sigma}(+\mathtt{a}) = \mathtt{L}j, \sigma; +\mathtt{a}; \#\#\mathtt{Lf_target}(1, (\mathtt{j}, \sigma), \rho); \#\#\mathtt{Lf_target}(2, (\mathtt{j}, \sigma), \rho),\\
&\vartheta^{\rho}_{j,\sigma}(-\mathtt{a}) = \mathtt{L}j, \sigma; -\mathtt{a}; \#\#\mathtt{Lf_target}(1, (\mathtt{j}, \sigma), \rho); \#\#\mathtt{Lf_target}(2, (\mathtt{j}, \sigma), \rho),\\
&\vartheta^{\rho}_{j,\sigma}(\mathbf{u}(X)) = \mathtt{L}j, \sigma; \mathtt{pglbu2pglbg}^{k',\rho}_{j,\sigma}(X)' \text{ where } k' \text{ is the length of } X,\\
&\vartheta^{\rho}_{j,\sigma}(u) = \mathtt{L}j, \sigma; u \text{ otherwise,}
\end{aligned}$$

and where the auxiliary functions

$$\begin{aligned}
&\mathtt{f_target} : \mathbb{N} \times \mathbb{N}^* \setminus \{\epsilon\} \times \mathbb{N}^* \to \mathbb{N}^* \setminus \{\epsilon\},\\
&\mathtt{b_target} : \mathbb{N} \times \mathbb{N}^* \setminus \{\epsilon\} \to \mathbb{N}^* \setminus \{\epsilon\}
\end{aligned}$$

are defined by:

$$\begin{aligned}
&\mathtt{f_target}(l, j, \epsilon) = l + j,\\
&\mathtt{f_target}(l, (j, j', \sigma), (k, \rho)) = \begin{cases} (l + j), j', \sigma & \text{if } l + j \leqslant k,\\ \mathtt{f_target}(l + j - k, (j', \sigma), \rho) & \text{otherwise.} \end{cases}
\end{aligned}$$

(Explanation of the last clause: there are $k - j$ steps possible on level j, j', σ, so $l + j - k$ are to be done on level j', σ.)

$$\begin{aligned}
&\mathtt{b_target}(l, j) = \max(0, j - l),\\
&\mathtt{b_target}(l, (j, j', \sigma)) = \begin{cases} (j - l), j', \sigma & \text{if } j - l \geqslant 1,\\ \mathtt{b_target}(l - j + 1, (j', \sigma)) & \text{otherwise.} \end{cases}
\end{aligned}$$

(Explanation of the last clause: on level j, j', σ there are $j - 1$ steps possible, so $l - j + 1$ remain on level j', σ.)

$$
\begin{aligned}
&\mathtt{pglbu2pglbg}^{\epsilon}_{\epsilon}(\ +\mathtt{a};\mathbf{u}(\mathtt{b};\backslash\#1;\backslash\#1);\mathtt{c};\backslash\#1;\backslash\#1\ )\\
&\qquad =\\
&\qquad\quad \mathtt{L}\,1;+\mathtt{a};\#\#\mathtt{L}\,\mathtt{f\_target}(1,1,\epsilon);\#\#\mathtt{L}\,\mathtt{f\_target}(2,1,\epsilon);\\
&\qquad\quad \mathtt{L}\,2;\mathtt{pglbu2pglbg}^{3}_{2}(\mathtt{b};\backslash\#1;\backslash\#1);\\
&\qquad\quad \mathtt{L}\,3;\mathtt{c};\\
&\qquad\quad \mathtt{L}\,4;\#\#\mathtt{L}\,\mathtt{b\_target}(1,4);\\
&\qquad\quad \mathtt{L}\,5;\#\#\mathtt{L}\,\mathtt{b\_target}(1,5)\\
&\qquad =\\
&\qquad\quad \mathtt{L}\,1;+\mathtt{a};\#\#\mathtt{L}\,2;\#\#\mathtt{L}\,3;\\
&\qquad\quad \mathtt{L}\,2;\mathtt{L}\,1,2;\mathtt{b};\\
&\qquad\qquad \mathtt{L}\,2,2;\#\#\mathtt{L}\,1,2;\\
&\qquad\qquad \mathtt{L}\,3,2;\#\#\mathtt{L}\,2,2;\\
&\qquad\quad \mathtt{L}\,3;\mathtt{c};\\
&\qquad\quad \mathtt{L}\,4;\#\#\mathtt{L}\,3;\\
&\qquad\quad \mathtt{L}\,5;\#\#\mathtt{L}\,4.
\end{aligned}
$$

Fig. 1. First example (continued) on pglbu2pglbg.

Note that if one starts from programs in first canonical form, the last clause of b_target $(l,(j,j',\sigma))$ is redundant. In Figs. 1 and 2, pglbu2pglbg is applied to the examples previously described.

5.3. *Projecting PGLB$_{\mathrm{g}}$ into PGLB*

The projection pglbg2pglb is defined by

$$\mathtt{pglbg2pglb}(u_1;\ldots,u_k)=\psi_1(u_1);\ldots;\psi_k(u_k),$$

where the auxiliary operation ψ_j is defined as follows:

$$\psi_j(\#\#\mathtt{L}\sigma)=\begin{cases}\#n & \text{if the leftmost occurrence of } \mathtt{L}\sigma \text{ is } n \text{ instructions forward,}\\ \backslash\#n & \text{if the leftmost occurrence of } \mathtt{L}\sigma \text{ is } n \text{ instructions backward,}\\ \#0 & \text{otherwise,}\end{cases}$$

$$\psi_j(\mathtt{L}\sigma)=\#1,$$

$$\psi_j(u)=u \text{ otherwise.}$$

In Fig. 3, pglbg2pglb is applied to the examples previously described.

$$\mathtt{pglbu2pglbg}^{\epsilon}_{\epsilon}(\ +\mathtt{a};\mathbf{u}(+\mathtt{b};\#5;!;\mathtt{c});\mathtt{d};+\mathtt{e};\backslash\#4;\backslash\#4;\backslash\#4;\backslash\#4;\backslash\#4\ )$$

$$=$$

$$\mathtt{L}\,1;+\mathtt{a};\#\#\mathtt{L}\,\mathtt{f\_target}(1,1,\epsilon);\#\#\mathtt{L}\,\mathtt{f\_target}(2,1,\epsilon);$$
$$\mathtt{L}\,2;\mathtt{pglbu2pglbg}^{4}_{2}(+\mathtt{b};\#5;!;\mathtt{c});$$
$$\mathtt{L}\,3;\mathtt{d};$$
$$\mathtt{L}\,4;+\mathtt{e};\#\#\mathtt{L}\,\mathtt{f\_target}(1,4,\epsilon);\#\#\mathtt{L}\,\mathtt{f\_target}(2,4,\epsilon);$$
$$\mathtt{L}\,5;\#\#\mathtt{L}\,\mathtt{b\_target}(4,5);$$
$$\mathtt{L}\,6;\#\#\mathtt{L}\,\mathtt{b\_target}(4,6)$$
$$\mathtt{L}\,7;\#\#\mathtt{L}\,\mathtt{b\_target}(4,7)$$
$$\mathtt{L}\,8;\#\#\mathtt{L}\,\mathtt{b\_target}(4,8)$$
$$\mathtt{L}\,9;\#\#\mathtt{L}\,\mathtt{b\_target}(4,9)$$

$$=$$

$$\mathtt{L}\,1;+\mathtt{a};\#\#\mathtt{L}\,2;\#\#\mathtt{L}\,3;$$
$$\mathtt{L}\,2;\mathtt{L}\,1,2;+\mathtt{b};\#\#\mathtt{L}\,2,2;\#\#\mathtt{L}\,3,2;$$
$$\mathtt{L}\,2,2;\#\#\mathtt{L}\,5;$$
$$\mathtt{L}\,3,2;!;$$
$$\mathtt{L}\,4,2;\mathtt{c};$$
$$\mathtt{L}\,3;\mathtt{d};$$
$$\mathtt{L}\,4;+\mathtt{e};\#\#\mathtt{L}\,5;\#\#\mathtt{L}\,6;$$
$$\mathtt{L}\,5;\#\#\mathtt{L}\,1;$$
$$\mathtt{L}\,6;\#\#\mathtt{L}\,2;$$
$$\mathtt{L}\,7;\#\#\mathtt{L}\,3;$$
$$\mathtt{L}\,8;\#\#\mathtt{L}\,4;$$
$$\mathtt{L}\,9;\#\#\mathtt{L}\,5.$$

Fig. 2. Second example (continued) on pglbu2pglbg.

5.4. Projecting PGLB into PGA

The projection pglb2pga is defined in [1], and reads

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

```
pglbg2pglb( L 1; +a; ##L 2; ##L 3;      =   #1; +a; #2; #8;
            L 2; L 1, 2; b;                 #1; #1; b;
                L 2, 2; ##L 1, 2;               #1; \#3;
                L 3, 2; ##L 2, 2;               #1; \#3;
            L 3; c;                         #1; c;
            L 4; ##L 3;                     #1; \#3;
            L 5; ##L 4 )                    #1; \#3.
```

```
pglbg2pglb( L 1; +a; ##L 2; ##L 3;                  =   #1; +a; #2; #12;
            L 2; L 1, 2; +b; ##L 2, 2; ##L 3, 2;        #1; #1; +b; #2; #3;
                L 2, 2; ##L 5;                              #1; #11;
                L 3, 2; !;                                  #1; !;
                L 4, 2; c;                                  #1; c;
            L 3; d;                                     #1; d;
            L 4; +e; ##L 5; ##L 6;                      #1; +e; #2; #3;
            L 5; ##L 1;                                 #1; \#22;
            L 6; ##L 2;                                 #1; \#20;
            L 7; ##L 3;                                 #1; \#11;
            L 8; ##L 4;                                 #1; \#11;
            L 9; ##L 5 )                                #1; \#9.
```

Fig. 3. Examples (continued) on `pglbg2pglb`.

where the auxiliary operation ψ_j is defined by:

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

In Fig. 4, `pglb2pga` is applied to the examples previously described. Extracting the behavior of the uppermost PGA program yields $|\mathtt{b}^\omega| \trianglelefteq \mathtt{a} \trianglerighteq |\mathtt{c}^\omega|$, where $|\mathtt{b}^\omega|$ abbreviates $\mathtt{b} \circ \mathtt{b} \circ \mathtt{b} \circ \cdots$, or equivalently, $|\mathtt{b}^\omega| = \mathtt{b} \circ |\mathtt{b}^\omega|$. This behavior equals (3), the behavior that was extracted with lazy projection in Section 4.1.

Extracting the behavior of the second PGA program, say X, is a tedious exercise. With unfolding the following characterization can be found:

$$|X| = (|X| \trianglelefteq \mathtt{b} \trianglerighteq \mathtt{S}) \trianglelefteq \mathtt{a} \trianglerighteq \mathtt{d} \circ (|X| \trianglelefteq \mathtt{e} \trianglerighteq (|X| \trianglelefteq \mathtt{b} \trianglerighteq \mathtt{S})).$$

```
pglb2pga( #1;+a;#2;#8;                    (k=17)=  ( #1;+a;#2;#8;
          #1;#1;b;#1;\#3;#1;\#3;                     #1;#1;b;#1;#16;#1;#16;
          #1;c;                                      #1;c;
          #1;\#3;                                    #1;#16;
          #1;\#3 )                                   #1;#16;
                                                                #0;#0 )^ω.
```

```
pglb2pga( #1;+a;#2;#12;                   (k=31)=  ( #1;+a;#2;#12;
          #1;#1;+b;#2;#3;                            #1;#1;+b;#2;#3;
              #1;#11;                                    #1;#11;
              #1;!;                                      #1;!;
              #1;c;                                      #1;c;
          #1;d;                                      #1;d;
          #1;+e;#2;#3;                               #1;+e;#2;#3;
          #1;\#22;                                   #1;#11;
          #1;\#20;                                   #1;#13;
          #1;\#11;                                   #1;#22;
          #1;\#11;                                   #1;#22;
          #1;\#9 )                                   #1;#24;
                                                              #0;#0 )^ω .
```

Fig. 4. Examples (continued) on `pglb2pga`.

This exactly matches the behavior of the originating $\mathrm{PGA_u}$ program (4), i.e.,

$$(+a;\mathbf{u}(+b;\#5;!;c);d;+e)^{\omega},$$

that was expected in Section 4.1 (cf. characterization (5) in that section).

6. Conclusion and digression

In this paper a projection from $\mathrm{PGA_u}$, i.e., PGA with unit instruction operators, into PGA is described in detail. The resulting projection `pgau2pga` is a functional composition, in which a projection from $\mathrm{PGLB_u}$ into $\mathrm{PGLB_g}$ constitutes the algorithmic kernel. This approach is chosen because the absence of repetitions seems to allow for a simpler type of bookkeeping. The latter projection is composed with an embedding `pgau2pgbu` and the appropriate projections into PGLB and PGA, respectively. It should be noticed that the embedding `pgau2pgbu` when restricted to PGA differs from `pga2pglb` as defined in [1,2]. The possible advantage of the present definition is that the instructions themselves need not

be transformed; only a sequence of backward jumps is added (and the possible occurrence of a repetition is omitted). Finally, it can be concluded that projections for the *relevant* PGA programming notations with unit instruction operators, i.e. $\mathrm{PGA}_{\mathrm{u}}$ and $\mathrm{PGLB}_{\mathrm{u}}$, are covered in this paper: for the program notations with more advanced programming features (see this issue) there is no reason to add a unit instruction operation, as its effect can easily be mimicked.

This paper is ended with a brief discussion of some topics that relate to the unit instruction operator:
(1) Equations for instruction sequence congruence and lazy projection of $\mathrm{PGA}_{\mathrm{u}}$.
(2) Second canonical forms for $\mathrm{PGA}_{\mathrm{u}}$.
(3) Composed instructions.
(4) Bisimulation equivalence.

Lazy projection was already mentioned in Section 4.1. The second topic discusses a refinement of the first canonical form for $\mathrm{PGA}_{\mathrm{u}}$ for which the projection into PGA yields in some cases much more concise programs. *Composed instructions*, i.e., propositional combinations of basic instructions as may occur in conditions in imperative programming languages, comprise an example of the unit instruction operator, as these can be simply rewritten into $\mathrm{PGA}_{\mathrm{u}}$ programs. Finally, *bisimulation equivalence* on $\mathrm{PGA}_{\mathrm{u}}$ programs coincides with behavioral equivalence and can be decided in polynomial time. Therefore, the equivalence of conditions with a side effect as may occur in imperative programming languages, can be decided in polynomial time.

6.1. Equations and lazy projection for the unit instruction operator

First, observe that the following equations are valid in the setting of instruction sequence congruence:

$$\mathbf{u}(u) = u,$$
$$\mathbf{u}(\mathbf{u}(X)) = \mathbf{u}(X),$$
$$\mathbf{u}(\mathbf{u}(X); Y) = \mathbf{u}(X; Y).$$

These equations can (of course) be used to remove occurrences of the unit instruction operator, thus allowing a more efficient projection of $\mathrm{PGA}_{\mathrm{u}}$ into PGA.

Next, in [1] it is stated that the semantic equations for $\mathrm{PGA}_{\mathrm{u}}$ satisfy

$$|\mathbf{u}(X)| = |X|, \quad |\mathbf{u}(X); Y| = |X; Y|.$$

Note that the first equation follows from the second one and the equation $|X| = |X; (\#0)^{\omega}|$ (the latter equation is present in [2]). These equations were given the characterization *lazy projection* in Section 4.1 and in [2], as opposed to the global or full projection `pgau2pga` defined in Section 5. It remains to be shown that lazy projection matches global projection for $\mathrm{PGA}_{\mathrm{u}}$. Of course, the equations for lazy projection are so natural that one might give these the status of a definition. In that perspective it remains to be shown that the projection `pgau2pga` is correct, i.e., $|\mathtt{pgau2pga}(X)| = |X|$.

6.2. Towards a second canonical form for PGA_{u}

A PGA program is in *second canonical form* if it is in first canonical form and satisfies the following two requirements:
(1) There are no chained jumps (i.e., subsequences of the form $\#n + 1; u_1; \ldots; u_n; \#m$).

(2) Counters used for a jump into the repeating part are as short as possible.

In [1,2] a transformation from first canonical form to second canonical form is described. The congruence that respects this transformation is called *structural congruence*, notation $=_{\mathrm{sc}}$, and properly includes instruction sequence congruence. For example,

$$\#2;\mathtt{a};\#3;\mathtt{b} =_{\mathrm{sc}} \#5;\mathtt{a};\#3;\mathtt{b},$$
$$\mathtt{a};\#9;(+\mathtt{b};!;\mathtt{c})^{\omega} =_{\mathrm{sc}} \mathtt{a};\#3;(+\mathtt{b};!;\mathtt{c})^{\omega}.$$

One can extend the definition of the second canonical form as described in [1,2] to $\mathrm{PGA_u}$ in the same way as was done with the first canonical form, except for the additional requirement that *all* jumps are "as short as possible". This refers to the situation where a jump exceeds the scope of a unit. The motivation to do so is that in some cases projecting a second canonical form yields a much more concise projection into PGA than the related first canonical form. The program

$$(+\mathtt{a};\mathbf{u}(+\mathtt{b};\#13;!;\mathtt{c});\mathtt{d};+\mathtt{e})^{\omega}$$

is not in second canonical form, as #13 can be minimized while preserving "structural congruence": the annotation

$$(+\mathtt{a};\ \overset{13}{\underset{5}{\overset{9}{}}}\ \mathbf{u}(+\mathtt{b};\#13;\ \overset{10}{6}\ \overset{1}{!};\ \overset{2}{\mathtt{c}});\ \overset{11}{\underset{3}{\overset{7}{\mathtt{d}}}};\ \overset{12}{\underset{4}{\overset{8}{+\mathtt{e}}}})^{\omega}$$

clarifies that

$$(+\mathtt{a};\mathbf{u}(+\mathtt{b};\#5;!;\mathtt{c});\mathtt{d};+\mathtt{e})^{\omega} \tag{6}$$

is in second canonical form. Clearly, pgau2pga projects the latter program to a much more concise PGA program than the former one. Obviously, the jump #k in the skeleton $(+\mathtt{a};\mathbf{u}(+\mathtt{b};\#k;!;\mathtt{c});\mathtt{d};+\mathtt{e})^{\omega}$ is minimal if $k \leqslant 6$. Of course, in the case of nested units the transformation into second canonical form is less simple, and is not considered here.

6.3. Composed instructions

In this section propositional composition of basic PGA instructions is introduced. Projecting composed instructions into PGA can be done with help of the unit instruction operator in a natural way, and thus provides an application of this operator. Composed instructions are built from Σ in the following way (ϕ, ψ ranging over composed instructions):

Negation. The composed instruction $\neg\phi$ has the same atomic behavior (sequence of actions) as ϕ and produces the negation of what ϕ produces.

Left-sequential conjunction. The composed instruction $\phi \mathbin{\diamond\!\!\wedge} \psi$ produces the result of ψ if ϕ produces true and then has the same atomic behavior as $\phi;\psi$, otherwise it produces false while behaving as ϕ.

Left-sequential disjunction. The composed instruction $\phi \mathbin{\diamond\!\!\vee} \psi$ produces true if ϕ does so and has the same atomic behavior as ϕ in that case, and otherwise it produces the result of ψ while behaving as $\phi;\psi$.

Composed instructions may be turned into composed test instructions by the prefix $+$ or $-$. Furthermore, composed instructions are projected into $\mathrm{PGA_u}$ as follows:

$$\phi \mapsto \mathbf{u}(+\phi;\#1),$$

$$-\phi \mapsto \mathbf{u}(+\phi; \#2),$$
$$+\neg\phi \mapsto -\phi,$$
$$+(\phi \mathbin{\diamond\!\wedge} \psi) \mapsto \mathbf{u}(-\phi; \#3; +\psi),$$
$$+(\phi \mathbin{\diamond\!\vee} \psi) \mapsto \mathbf{u}(+\phi; \#2; +\psi).$$

As an example, set $\phi = +(\neg a \mathbin{\diamond\!\wedge} b); c$ and $\psi = +\neg(a \mathbin{\diamond\!\vee} \neg b); c$. Then

$$|\phi| = |\mathbf{u}(\mathbf{u}(-\mathtt{a}; \#2); \#3; +\mathtt{b}); \mathtt{c}| = |\mathbf{u}(-\mathtt{a}; \#2; \#3; +\mathtt{b}); \mathtt{c}|,$$
$$|\psi| = |\mathbf{u}(\mathbf{u}(+\mathtt{a}; \#2; -\mathtt{b}); \#2); \mathtt{c}| = |\mathbf{u}(+\mathtt{a}; \#2; -\mathtt{b}; \#2); \mathtt{c}|,$$

which indeed are the same:

a	b	$\lvert\phi\rvert$	$\lvert\psi\rvert$
true	true	a ∘ D	a ∘ D
true	false	a ∘ D	a ∘ D
false	true	a ∘ b ∘ c ∘ D	a ∘ b ∘ c ∘ D
false	false	a ∘ b ∘ D	a ∘ b ∘ D

Composed instructions can be regarded in a fixed context, e.g. in the template

$$R_\phi = +\phi; \#3; \mathtt{WF}; !; \mathtt{WT}; !$$

with ϕ a composed instruction. R_ϕ produces the observable action WT ("write true") in case ϕ yields true, and the observable action WF ("write false") otherwise. Now $R_\phi \equiv_{\mathrm{be}} R_\psi$ if and only if $\phi \equiv_{\mathrm{be}} \psi$. The above shows that in principle there exists a procedure for deciding $\phi \equiv_{\mathrm{be}} \psi$ (namely, via the projection pgau2pga). Furthermore, this can be decided in polynomial time by employing *bisimulation equivalence* as discussed below. As a consequence, the equivalence of propositions with a side effect as may occur in conditions in programming languages as C [5] or Java [3], is decidable in polynomial time.

6.4. Bisimulation equivalence

One can define a version of bisimulation equivalence [6] that identifies two programs whenever they give rise to step-wise similar behavior. Below this equivalence is sketched for PGA programs of the form

$$X; Y^\omega,$$

where X and Y do not contain repetition. In terms of behavior, this particular variant of the first canonical form is not a restriction by the identification

$$|X| = |X; (\#0)^\omega|,$$

or, $X \equiv_{\mathrm{be}} X; (\#0)^\omega$. Now two programs X and Y of the above form are *bisimilar*, notation

$$X \sim Y$$

if there exists a binary relation B that relates the "instruction positions" of X with those of Y such that $(1, 1) \in B$, and whenever $(i, j) \in B$ and the ith instruction of X gives rise to an

atomic behavior, then the jth instruction of Y should match this behavior and vice versa, and the resulting instruction positions should again be in B. Such a relation is then called a *bisimulation*. For example,

$$\mathtt{a};(+\mathtt{b})^{\omega} \sim +\mathtt{a};\mathtt{b}^{\omega}$$

are related by the bisimulation $\{(1,1),(2,2)\}$ and for a less trivial example,

$$\mathtt{a};(\mathtt{b};+\mathtt{c})^{\omega} \sim \mathtt{a};\#2;(\#1;\mathtt{b};+\mathtt{c};\#7;\#8;\mathtt{d};\mathtt{b};+\mathtt{c};\#2;\#3)^{\omega}. \tag{7}$$

As to the latter example, annotate both programs with instruction positions:

$$\mathtt{a}_1;(\mathtt{b}_2;(+\mathtt{c})_3)^{\omega},$$
$$\mathtt{a}_1;(\#2)_2;((\#1)_3;\mathtt{b}_4;(+\mathtt{c})_5;(\#7)_6;(\#8)_7;\mathtt{d}_8;\mathtt{b}_9;(+\mathtt{c})_{10};(\#2)_{11};(\#3)_{12})^{\omega}.$$

A bisimulation that witnesses (7) is the relation R defined by

$$R=\{(1,1),(2,2),(2,3),(2,4),(2,6),(3,5),(3,7)\}.$$

Another bisimulation that also witnesses (7) is

$$R'=R\cup\{(2,9),(2,11),(3,10),(3,12)\}.$$

For PGA programs, a bisimulation relation as described above can be defined using a large number of case distinctions. For instance, in the following style: for programs

$$X=u_1;\ldots;u_k;(u_{k+1};\ldots;u_{k+n})^{\omega} \quad \text{and} \quad Y=v_1;\ldots;v_l;(v_{l+1};\ldots;v_{l+m})^{\omega},$$

one can define addition $\oplus_X$ and $\oplus_Y$ in such a way that the length of the repeating part of X, respectively Y, is balanced with the newly computed position. Then, if $(i,j)\in B$ and $u_i=\mathtt{a}$, then either $v_j=\mathtt{a}$ and $(i\oplus_X 1, j\oplus_Y 1)\in B$, or $v_j\in\{+\mathtt{a},-\mathtt{a}\}$ and $\{(i\oplus_X 1, j\oplus_Y 1),(i\oplus_X 1, j\oplus_Y 2)\}\subseteq B$, or $v_j=\#n'+1$ and $(i, j\oplus_Y (n'+1))\in B$. For the cases that $u_i=+\mathtt{a}, -\mathtt{a}, !$ and $\#n'$, similar requirements are needed, as well as for the symmetric cases that start from the form of v_j.

It should be clear that bisimulation equivalence coincides with behavioral equivalence. Furthermore, bisimulation equivalence can be decided in polynomial time (see, e.g., [4] on the complexity of bisimilarity for finite structures). Because the projection `pgau2pga` is polynomial, as well as the necessary preprocessing of $\mathrm{PGA_u}$ programs into first canonical form, this establishes the claim made in the previous section: the equivalence of propositions with a side effect as may occur in conditions in programming languages as C [5] or Java [3], is decidable in polynomial time.

Acknowledgements

I thank Jan Bergstra for providing the example in which composed instructions are introduced (Section 6.3), and Inge Bethke and a referee for useful comments.

References

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

[2] J.A. Bergstra, M.E. Loots, Program algebra for sequential code, J. Logic Algebr. Programming 51 (2002) 125–156.

[3] G. Bracha, J. Gosling, B. Joy, G. Steele, The Java Language Specification, second ed., Addison-Wesley, New York, 2000 (First edition by J. Gosling, B. Joy, G. Steele, 1996).
[4] R. Cleaveland, O. Sokolsky, Equivalence and preorder checking for finite-state systems, in: J.A. Bergstra, A. Ponse, S.A. Smolka (Eds.), Handbook of Process Algebra, Elsevier, Amsterdam, 2001, pp. 391–424.
[5] B.W. Kernighan, D.M. Ritchie, The C Programming Language, second ed., Prentice-Hall, Englewood Cliffs, NJ, 1988.
[6] D.M.R. Park, Concurrency and automata on infinite sequences, in: P. Deussen (Ed.), Proceedings of the 5th GI (Gesellschaft für Informatik) Conference, Karlsruhe, Lecture Notes in Computer Science, vol. 104, Springer, Berlin, 1981, pp. 167–183.