Polarized Process Algebra and Program Equivalence

Jan A. Bergstra[1,2] and Inge Bethke[2]

[1] Applied Logic Group, Department of Philosophy, Utrecht University, Heidelberglaan 8, 3584 CS Utrecht, The Netherlands, Jan.Bergstra@phil.uu.nl
[2] Programming Research Group, Informatics Institute, Faculty of Science, University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands, inge@science.uva.nl

Abstract. The basic polarized process algebra is completed yielding as a projective limit a cpo which also comprises infinite processes. It is shown that this model serves in a natural way as a semantics for several program algebras. In particular, the fully abstract model of the program algebra axioms of [2] is considered which results by working modulo behavioral congruence. This algebra is extended with a new basic instruction, named 'entry instruction' and denoted with '@'. Addition of @ allows many more equations and conditional equations to be stated. It becomes possible to find an axiomatization of program inequality. Technically this axiomatization is an infinite final algebra specification using conditional equations and auxiliary objects.

1 Introduction

Program algebra as introduced in [2] and [3] is a tool for the conceptualization of programs and programming. It is assumed that a program is executed in a context composed of components complementary to the program. While a program's actions constitute requests to be processed by an environment, the complementary system components in an environment view actions as request issued by another party (the program being run). After each request the environment may undergo a state change whereupon it replies with a boolean value. The boolean return value is used to decide how the execution of the program will continue.

For theoretical work on program algebra a semantic model is important. It is assumed that the meaning of a program is a process. A particular kind of processes termed *polarized processes* is well-suited to serve as the semantic interpretation of a program. In this paper the semantic world of polarized processes is introduced following the presentation of [3]. Polarized process algebra can stand on its own feet though significant results allowing to maintain it as an independent subject are currently missing. Then program algebra is introduced as a formalism for denoting objects (programs) that can be mapped into the set of polarized processes in a natural fashion. Several program algebras are defined. One of these structures may be classified as fully abstract. The focus

J.C.M. Baeten et al. (Eds.): ICALP 2003, LNCS 2719, pp. 1–21, 2003.

of the paper is on an analysis of aspects of that model. This eventually leads to a final algebra specification of the fully abstract model. It seems to be the case that the fully abstract program algebra resists straightforward methods of algebraic specification. No negative results have been obtained, however. Several problems are left open.

2 Basic Polarized Process Algebra

Most process algebras (e.g. ACP from [1] and TCSP from [6]) are non-polarized. This means that in a parallel composition of process P and Q, both processes and their actions have a symmetric status. In a polarized setting each action has a definite asymmetric status. Either it is a request or it is (part of) the processing of a request. When a request action is processed a boolean value is returned to the process issuing the request. When this boolean value is returned the processing of the request is completed.

Non-polarized process algebra may be (but need not) considered the simplified case in which always `true` is returned. Polarized process algebra is less elegant than non-polarized process algebra. Its advantage lies in the more direct modeling of sequential deterministic systems. Polarized process algebra need not dive into the depths of choice and non-determinism when deterministic systems are discussed.

BPPA is based on a collection Σ of basic actions[1]. Each action is supposed to be polarized and to produce a boolean value when executed. In addition its execution may have some side-effect in an environment. One imagines the boolean value mentioned above to be generated while this side-effect on the environment is being produced. BPPA has two constants which are meant to model termination and inaction and two composition mechanisms, the second one of these being defined in terms of the first one.

Definition 1. *For a collection Σ of atomic actions, $BPPA_\Sigma$ denotes the family of processes inductively defined by*

termination: $S \in BPPA_\Sigma$

With S (stop) terminating behavior is denoted; it does no more than terminate. Termination actions will not have any side effect on a state.

inaction: $D \in BPPA_\Sigma$

By D (sometimes just 'loop') an inactive behavior is indicated. It is a behav-

[1] The phrase 'basic action' is used in polarized process algebra in contrast with 'atomic action' as used in process algebra. Indeed from the point of view of ordinary process algebra the basic actions are not considered atomic. In program algebra the phrase 'basic instruction' is used. Basic instructions are mapped on basic actions if the semantics of program algebra is described in terms of a polarized process algebra. Program algebra also features so-called primitive instructions. These are the basic instructions without test (void uses) and with positive or negative test, the termination instruction as well as a jump instruction $\#n$ for each $n \in \mathbb{N}$.

ior that represents the impossibility of making real progress, for instance an internal cycle of activity without any external effect whatsoever[2].

postconditional composition: *For action $a \in \Sigma$ and processes P and Q in $BPPA_\Sigma$*

$$P \trianglelefteq a \trianglerighteq Q \in BPPA_\Sigma$$

This composition mechanism denotes the behavior that first performs a and then either proceeds with P if `true` *was produced or with Q otherwise.*

For $a \in \Sigma$ and process $P \in BPPA_\Sigma$, we abbreviate the postconditional composition $P \trianglelefteq a \trianglerighteq P$ by

$$a \circ P$$

and call this composition mechanism **action prefix.**

Thus all processes in BPPA_Σ are made from S and D by means of a finite number of applications of postconditional composition. This suggests the existence of a partial ordering and an operator which finitely approximates every basic process.

Definition 2. *1. Let $\sqsubseteq$ be the partial ordering on $BPPA_\Sigma$ generated by the clauses*

a) for all $P \in BPPA_\Sigma$, $D \sqsubseteq P$, and
b) for all $P, Q, X, Y \in BPPA_\Sigma$, $a \in \Sigma$,

$$P \sqsubseteq X \;\&\; Q \sqsubseteq Y \Rightarrow P \trianglelefteq a \trianglerighteq Q \sqsubseteq X \trianglelefteq a \trianglerighteq Y.$$

2. Let $\pi : \mathbb{N} \times BPPA_\Sigma \to BPPA_\Sigma$ be the approximation operator determined by the equations

a) for all $P \in BPPA_\Sigma$, $\pi(0, P) = D$,
b) for all $n \in \mathbb{N}$, $\pi(n+1, S) = S$, $\pi(n+1, D) = D$, and
c) for all $P, Q \in BPPA_\Sigma, n \in \mathbb{N}$,

$$\pi(n+1, P \trianglelefteq a \trianglerighteq Q) = \pi(n, P) \trianglelefteq a \trianglerighteq \pi(n, Q).$$

We shall write $\pi_n(P)$ instead of $\pi(n, P)$.

π finitely approximates every process in BPPA_Σ. That is,

Proposition 1. *For all $P \in BPPA_\Sigma$,*

$$\exists n \in \mathbb{N}\ \pi_0(P) \sqsubseteq \pi_1(P) \sqsubseteq \cdots \sqsubseteq \pi_n(P) = \pi_{n+1}(P) = \cdots = P.$$

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

Proof. We employ structural induction. If $P = D$ or $P = S$ then n can be taken 0 or 1, respectively. If $P = P_1 \trianglelefteq a \trianglerighteq P_2$ let $n, m \in \mathbb{N}$ be such that $\pi_0(P_1) \sqsubseteq \pi_1(P_1) \sqsubseteq \cdots \sqsubseteq \pi_n(P_1) = \pi_{n+1}(P_1) = \cdots = P_1$ and $\pi_0(P_2) \sqsubseteq \pi_1(P_2) \sqsubseteq \cdots \sqsubseteq \pi_m(P_2) = \pi_{m+1}(P_2) = \cdots = P_2$. Thus for $k = max\{n, m\}$ we have

$$\begin{aligned}
&\pi_0(P_1) \trianglelefteq a \trianglerighteq \pi_0(P_2) \sqsubseteq \pi_1(P_1) \trianglelefteq a \trianglerighteq \pi_1(P_2) \\
&\qquad\vdots \\
&\sqsubseteq \pi_k(P_1) \trianglelefteq a \trianglerighteq \pi_k(P_2) \\
&= \pi_{k+1}(P_1) \trianglelefteq a \trianglerighteq \pi_{k+1}(P_2) \\
&\qquad\vdots \\
&= P_1 \trianglelefteq a \trianglerighteq P_2.
\end{aligned}$$

Hence $\pi_0(P) \sqsubseteq \pi_1(P) \sqsubseteq \cdots \sqsubseteq \pi_{k+1}(P) = \pi_{k+2}(P) = \cdots = P$.

Polarized processes can be finite or infinite. Following the metric process theory of [7] in the form developed as the basis of the introduction of processes in [1], BPPA_Σ has a completion $\mathrm{BPPA}_\Sigma^\infty$ which comprises also the infinite processes. Standard properties of the completion technique yield that we may take $\mathrm{BPPA}_\Sigma^\infty$ as consisting of all so-called *projective* sequences.

Recall that a *directed* set is a non-empty, partially ordered set which contains for any pair of its elements an upper bound. A *complete partial order (cpo)* is a partially ordered set with a least element such that every directed subset has a supremum. Let $C_0, C_1, \ldots$ be a countable sequence of cpo's and let $f_i : C_{i+1} \to C_i$ be continuous for every $i \in \mathbb{N}$. The sequence (C_i, f_i) is called a *projective* (or *inverse*) system of cpo's. The *projective* (or *inverse*) *limit* of the system (C_i, f_i) is the poset $(C^\infty, \sqsubseteq)$ with

$$C^\infty = \{(x_i)_{i\in\mathbb{N}} \mid \forall i \in \mathbb{N}\ x_i \in C_i \ \&\ f_i(x_{i+1}) = x_i\}$$

and

$$(x_i)_{i\in\mathbb{N}} \sqsubseteq (y_i)_{i\in\mathbb{N}} \Leftrightarrow \forall i \in \mathbb{N}\ x_i \sqsubseteq y_i.$$

A fundamental theorem of domain theory states that C^∞ is a cpo with

$$\bigsqcup X = (\bigsqcup_{x\in X} x_i)_{i\in\mathbb{N}}$$

for directed $X \subseteq C^\infty$. If in addition there are continuous mappings $g_i : C_i \to C_{i+1}$ such that for every $i \in \mathbb{N}$

$$f_i(g_i(x)) = x \text{ and } g_i(f_i(x)) \sqsubseteq x$$

then, up to isomorphism, $C_i \subseteq C^\infty$. The isomorphism $h_i : C_i \to C^\infty$ can be given by

$$h_i(x) = \langle f_0(f_1 \cdots, f_{i-1}(x) \cdots), \cdots f_{i-1}(x), x, g_i(x), g_{i+1}(g_i(x)), \cdots \rangle.$$

Hence, up to isomorphism, $\bigcup_{i\in\mathbb{N}} C_i \subseteq C^\infty$. For a detailed account of this construction consult e.g. [11].

Definition 3.

1. *For all* $n \in \mathbb{N}$, $BPPA^n_\Sigma = \{\pi_n(P) \mid P \in BPPA_\Sigma\}$
2. $BPPA^\infty_\Sigma = \{(P_n)_{n\in\mathbb{N}} \mid \forall n \in \mathbb{N}(P_n \in BPPA^n_\Sigma \;\&\; \pi_n(P_{n+1}) = P_n)\}$

Lemma 1. *Let* $(C, \sqsubseteq)$ *be a finite directed set. Then* C *has a maximal element.*

Proof. Say $C = \{c_0, c_1, \ldots, c_n\}$. If $n = 0$, c_0 is maximal. Otherwise pick $x_0 \in C$ such that $c_0, c_1 \sqsubseteq x_0$ and for $1 \leq i \leq n-1$ pick $x_i \in C$ such that $x_{i-1}, c_{i+1} \sqsubseteq x_i$. $x_0, x_1, \ldots, x_{n-1}$ exist since C is directed. Now notice that x_{n-1} is the maximal element.

Proposition 2. *For all* $n \in \mathbb{N}$,

1. $BPPA^n_\Sigma$ *is a cpo,*
2. π_n *is continuous,*
3. *for all* $P \in BPPA_\Sigma$,
   a) $\pi_n(P) \sqsubseteq P$,
   b) $\pi_n(\pi_n(P)) = \pi_n(P)$, *and*
   c) $\pi_{n+1}(\pi_n(P)) = \pi_n(P)$.

Proof.

1. We prove by induction on n that every directed set $X \subseteq \mathrm{BPPA}^n_\Sigma$ is finite. It then follows from the previous lemma that suprema exist: they are the maximal elements. The base case is trivial since $\mathrm{BPPA}^0_\Sigma = \{D\}$. Now consider any directed $X \subseteq \mathrm{BPPA}^{n+1}_\Sigma$. We distinguish two cases.
   a) $S \in X$: Then $X \subseteq \{D, S\}$. Thus X is finite.
   b) $S \notin X$: Since X is directed there exists a unique $a \in \Sigma$ such that $X \subseteq \{D, \pi_n(P) \trianglelefteq a \trianglerighteq \pi_n(Q) \mid P, Q \in \mathrm{BPPA}_\Sigma\}$. Now let $X_1 = \{D, \pi_n(P) \mid \exists Q \in \mathrm{BPPA}_\Sigma\ \pi_n(P) \trianglelefteq a \trianglerighteq \pi_n(Q) \in X\}$ and $X_2 = \{D, \pi_n(Q) \mid \exists P \in \mathrm{BPPA}_\Sigma\ \pi_n(P) \trianglelefteq a \trianglerighteq \pi_n(Q) \in X\}$. Since X is directed it follows that both X_1 and X_2 are directed and hence finite by the induction hypothesis. Thus X is finite.
2. Since directed subsets are finite it suffices to show that π_n is monotone. Let $P \sqsubseteq Q \in \mathrm{BPPA}_\Sigma$. We employ again induction on n. π_0 is constant and thus monotone. For $n+1$ we distinguish three cases.
   a) $P = D$: Then $\pi_{n+1}(P) = D \sqsubseteq \pi_{n+1}(Q)$.
   b) $P = S$: Then also $Q = S$. Hence $\pi_{n+1}(P) = \pi_{n+1}(Q)$.
   c) $P = P_1 \trianglelefteq a \trianglerighteq P_2$: Then $Q = Q_1 \trianglelefteq a \trianglerighteq Q_2$ with $P_i \sqsubseteq Q_i$ for $i \in \{1, 2\}$. From the monotonicity of π_n it now follows that $\pi_n(P_i) \sqsubseteq \pi_n(Q_i)$ for $i \in \{1, 2\}$. Thus $\pi_{n+1}(P) \sqsubseteq \pi_{n+1}(Q)$.
3. Let $P \in BPPA_\Sigma$. (a) follows from Proposition 1. We prove (b) and (c) simultaneously by induction on n. For $n = 0$ we have $\pi_0(\pi_0(P)) = D = \pi_0(P)$ and $\pi_1(\pi_0(P)) = D = \pi_0(P)$. Now consider $n+1$. We distinguish two cases.

a) $P \in \{D, S\}$: Then $\pi_{n+1}(\pi_{n+1}(P)) = P = \pi_{n+1}(P)$ and $\pi_{n+2}(\pi_{n+1}(P)) = P = \pi_{n+1}(P)$.
b) $P = P_1 \trianglelefteq a \trianglerighteq P_2$: Then it follows from the induction hypothesis that

$$\begin{aligned}\pi_{n+1}(\pi_{n+1}(P)) &= \pi_n(\pi_n(P_1)) \trianglelefteq a \trianglerighteq \pi_n(\pi_n(P_2)) \\ &= \pi_n(P_1) \trianglelefteq a \trianglerighteq \pi(P_2) = \pi_{n+1}(P)\end{aligned}$$

and

$$\begin{aligned}\pi_{n+2}(\pi_{n+1}(P)) &= \pi_{n+1}(\pi_n(P_1)) \trianglelefteq a \trianglerighteq \pi_{n+1}(\pi_n(P_2)) \\ &= \pi_n(P_1) \trianglelefteq a \trianglerighteq \pi(P_2) = \pi_{n+1}(P).\end{aligned}$$

Theorem 1. *$BPPA^{\infty}_{\Sigma}$ is a cpo and, up to isomorphism, $BPPA_{\Sigma} \subseteq BPPA^{\infty}_{\Sigma}$.*

Proof. 1. and 2. of the previous proposition show that $(\mathrm{BPPA}^n_{\Sigma}, \pi_n)$ is a projective system of cpo's. Thus $\mathrm{BPPA}^{\infty}_{\Sigma}$ is a cpo. Note that it follows from 3(c) that $\mathrm{BPPA}^n_{\Sigma} \subseteq \mathrm{BPPA}^{n+1}_{\Sigma}$ for all n. Thus if we define for all P and n, $id_n(P) = P$ then $id_n : \mathrm{BPPA}^n_{\Sigma} \to \mathrm{BPPA}^{n+1}_{\Sigma}$ for all n. id_n is clearly continuous. Moreover, 3(a) yields $\pi_n(id_n(P)) \sqsubseteq P$ for all n and $P \in \mathrm{BPPA}^n_{\Sigma}$. Likewise, 3(b) yields $id_n(\pi_n(P)) = P$ for all n and $P \in \mathrm{BPPA}^{n+1}_{\Sigma}$. Thus, up to isomorphism, $\bigcup_{n\in\mathbb{N}} \mathrm{BPPA}^n_{\Sigma} \subseteq \mathrm{BPPA}^{\infty}_{\Sigma}$. Thus also $\mathrm{BPPA}_{\Sigma} \subseteq \mathrm{BPPA}^{\infty}_{\Sigma}$ since $\mathrm{BPPA}_{\Sigma} = \bigcup_n \mathrm{BPPA}^n_{\Sigma}$ by Proposition 1.

The set of polarized processes can serve in a natural fashion as a semantics for programs. As an example we shall consider PGA_{Σ}.

3 Program Algebra

Given a collection Σ of atomic instructions the syntax of program expressions (or programs) in PGA_{Σ} is generated from five kinds of constants and two composition mechanisms. The constants are made from Σ together with a termination instruction, two test instructions and a forward jump instruction. As in the case of BPPA, the atomic 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 two composition mechanisms are concatenation and infinite repetition.

Definition 4. *For a collection Σ of atomic instructions, PGA_{Σ} denotes the collection of program expressions inductively defined by*

termination: $! \in PGA_{\Sigma}$
The instruction ! indicates termination of the program and will not return any value.

forward jump instruction: $\#n \in PGA_{\Sigma}$ *for every* $n \in \mathbb{N}$
n counts how many subsequent instructions must be skipped, including the jump instruction itself.

void basic instruction: $a \in PGA_\Sigma$ *for every* $a \in \Sigma$
positive test instruction: $+a \in PGA_\Sigma$ *for every* $a \in \Sigma$
The execution of $+a$ *begins with executing* a*. Thereafter, if* **true** *is replied, program execution continues with the execution of the next instruction following the positive test instruction in the program. Otherwise, if* **false** *is replied, the instruction immediately following the (positive) test instruction is skipped and program execution continues with the instruction thereafter.*
negative test instruction: $-a \in PGA_\Sigma$ *for every* $a \in \Sigma$
The negative test instruction $(-a)$ *reacts the other way around on the boolean values it receives as a feedback from its operating context. At a positive (*__true__*) reply it skips the next action, and at a negative reply it simply continues.*
concatenation: *For programs* $X, Y \in PGA_\Sigma$, $X; Y \in PGA_\Sigma$
repetition: *For a program* $X \in PGA_\Sigma$, $X^\omega \in PGA_\Sigma$

Here are some program examples:

$$+a; !; +b; \#3; c; !; d; !$$

$$a; !; -b; \#3; c; \#0; d; !$$

$$-a; !; (-b; \#3; c; \#0; +d; !)^\omega.$$

The simplest model of the signature of program algebra interprets each term as a sequence of primitive instructions. This is the *instruction sequence model.* Equality within this model will be referred to as instruction sequence congruence ($=_{\mathrm{isc}}$). Two programs X and Y are instruction sequence congruent if both denote the same sequence of instructions after unfolding the repetition operator, that is, if they can be shown to be equal by means of the program object equations in Table 1.

Table 1. 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 $X^{n+1} = X; X^n$. The associativity of concatenation implies as usual that far fewer brackets have to be used. We will use associativity whenever confusion cannot emerge.

The program object equations allow some useful transformations, in particular the transformation into *first canonical form.*

Definition 5. *Let $X \in PGA_\Sigma$. Then X is in first canonical form iff*

1. *X does not contain any repetition, or*
2. *$X = Y; Z^\omega$ with Y and Z not containing any repetition.*

The existence of first canonical forms follows straightforwardly by structural induction. The key case is this:

$$\begin{aligned}(U; X^\omega)^\omega &=_{\mathrm{isc}} (U; X^\omega; U; X^\omega)^\omega && \text{by PGA2}\\ &=_{\mathrm{isc}} (U; X^\omega); (U; X^\omega)^\omega && \text{by PGA4}\\ &=_{\mathrm{isc}} U; (X^\omega; (U; X^\omega)^\omega) && \text{by PGA1}\\ &=_{\mathrm{isc}} U; X^\omega && \text{by PGA3}\end{aligned}$$

First canonical forms need not be unique. For example, $a; a; a^\omega$ and $a; a; a; a^\omega$ are both canonical forms of $a; a^\omega$ which is already in canonical form itself. In the sequel we shall mean by *the* first canonical form the shortest one.

Definition 6. *Let $X \in PGA_\Sigma$ be in first canonical form. The length of X, $l(X)$, is defined by*

1. *if X does not contain any repetition then $l(X) = (n, 0)$ where n is the number of instructions in X, and*
2. *if $X = Y; Z^\omega$ with both Y and Z not containing any repetition then $l(X) = (n, m)$ where n and m are the number of instructions in Y and Z, respectively.*

Observe that $\mathbb{N} \times \mathbb{N}$ is a well-founded partial order by stipulating

$$(n_0, n_1) \leq (m_0, m_1) \Leftrightarrow n_0 \leq m_0 \text{ or } (n_0 = m_0 \text{ and } n_1 \leq m_1).$$

Definition 7. *Let $X \in PGA_\Sigma$.* The first canonical form of X, *$cf(X)$, is a first canonical form X' with $X =_{\mathrm{isc}} X'$ and minimal length; i.e. for all first canonical forms X'' with $X =_{\mathrm{isc}} X''$, $l(X') \leq l(X'')$. We call X* finite *if $l(cf(X)) = (n, 0)$ and* infinite *if $l(cf(X)) = (n, m+1)$ for some $n, m \in \mathbb{N}$.*

Clearly $cf(X)$ is well-defined, that is, there exists a unique shortest first canonical form of X.

A second model of program algebra is $\mathrm{BPPA}^\infty_\Sigma$. As a prerequisite we define a mapping $|\ |$ from finite programs, i.e. programs without repetition, to finite polarized processes. Prior to a formal definition some examples are of use:

$$|a; b; !| = a \circ (b \circ S)$$

$$|a; +b; !; \#0| = a \circ (S \trianglelefteq b \trianglerighteq D)$$

$$|+a; !| = S \trianglelefteq a \trianglerighteq D.$$

The intuition behind the mapping to processes is as follows: view a program as an instruction sequence and turn that into a process from left to right. The mapping into processes removes all control aspects (tests, jumps) in favor of an unfolding of all possible behaviors. A forward jump instruction with counter zero jumps to itself, thereby creating a loop or divergence (D). Only via ! the proper termination (S) will take place. If the program is exited in another way this also counts as a divergence (D).

In the sequel we let $u, u_1, u_2, \ldots$ range over $\{!, \#k, a, +a, -a | a \in \Sigma, k \in N\}$.

Definition 8. *Let $X \in PGA_\Sigma$ be finite. Then $|X|$ is defined by induction on its length $l(X)$.*

1. $l(X) = (1,0)$:
   a) *If* $X =!$ *then* $|X| = S$,
   b) *if* $X = \#k$ *then* $|X| = D$, *and*
   c) *if* $X \in \{a, +a, -a\}$ *then* $|X| = a \circ D$.
2. $l(X) = (n+2, 0)$:
   a) *if* $X =!; Y$ *then* $|X| = S$,
   b) *if* $X = \#0; Y$ *then* $|X| = D$,
   c) *if* $X = \#1; Y$ *then* $|X| = |Y|$,
   d) *if* $X = \#k+2; u; Y$ *then* $|X| = |\#k+1; Y|$,
   e) *if* $X = a; Y$ *then* $|X| = a \circ |Y|$;
   f) *if* $X = +a; Y$ *then* $|X| = |Y| \trianglelefteq a \trianglerighteq |\#2; Y|$, *and*
   g) *if* $X = -a; Y$ *then* $|X| = |\#2; Y| \trianglelefteq a \trianglerighteq |Y|$.

Observe that $|\ |$ is monotone in continuations. That is,

Proposition 3. *Let $X = u_1; \cdots; u_n$ and $Y = u_1; \cdots; u_n; \cdots; u_{n+k}$. Then $|X| \sqsubseteq |Y|$.*

Proof. Straightforward by induction on n and case ramification. E.g. if $n = 1$ and $X \in \{a, +a, -a\}$ then $|X| = a \circ D$ and $|Y| = |Z| \trianglelefteq a \trianglerighteq |Z'|$ for some $Z, Z' \in \mathrm{PGA}_\Sigma$. Thus $|X| \sqsubseteq |Y|$. If $n > 1$ consider e.g. the case where $X = \#k+2; u_2; \cdots; u_n$. Then $|X| = |\#k+1; u_3; \cdots; u_n| \sqsubseteq |\#k+1; u_3; \cdots; u_n; \cdots; u_{n+k}| = |Y|$ by the induction hypothesis. Etc.

It follows that for repetition-free Y and Z, $|Y; Z| = |Y; Z^1| \sqsubseteq |Y; Z^2| \sqsubseteq |Y; Z^3| \sqsubseteq \cdots$ is an ω-chain and hence directed. Thus $\bigsqcup_{n \in \mathbb{N}} |Y; Z^n|$ exists in $\mathrm{BPPA}^\infty_\Sigma$. We can now extend Definition 8 to infinite processes.

Definition 9. *Let $Y; Z^\omega \in PGA_\Sigma$ be in first canonical form. Then $|Y; Z^\omega| = \bigsqcup_{n \in \mathbb{N}} |Y; Z^n|$.*

Moreover, for arbitrary programs we define

Definition 10.
Let $X \in PGA_\Sigma$. Then $[\![X]\!] = |cf(X)|$.

As an example consider:

$$\begin{aligned}
[\![+a; \#3; !; (b;c)^\omega]\!] &= \bigsqcup_{n\in\mathbb{N}} | +a; \#3; !; (b;c)^n | \\
&= \bigsqcup_{n\in\mathbb{N}} |\#3; !; (b;c)^n| \trianglelefteq a \trianglerighteq \bigsqcup_{n\in\mathbb{N}} |\#2; \#3; !; (b;c)^n| \\
&= \bigsqcup_{n\in\mathbb{N}} |\#2; (b;c)^n| \trianglelefteq a \trianglerighteq \bigsqcup_{n\in\mathbb{N}} |\#1; !; (b;c)^n| \\
&= \bigsqcup_{n\in\mathbb{N}} |\#1; (c;b)^n| \trianglelefteq a \trianglerighteq \bigsqcup_{n\in\mathbb{N}} |!; (b;c)^n| \\
&= \bigsqcup_{n\in\mathbb{N}} |(c;b)^n| \trianglelefteq a \trianglerighteq \bigsqcup_{n\in\mathbb{N}} |!; (b;c)^n| \\
&= c \circ b \circ c \circ b \circ \cdots \trianglelefteq a \trianglerighteq S
\end{aligned}$$

Since instruction sequence congruent programs have identical cf-canonical forms we have

Theorem 2. *For all $X, Y \in PGA_\Sigma$, $X =_{\mathrm{isc}} Y \Rightarrow [\![X]\!] = [\![Y]\!]$.*

The converse does not hold: e.g. $\#1;! \neq_{\mathrm{isc}} !$ but $[\![\#1;!]\!] = S = [\![!]\!]$.

Further models for program algebra will be found by imposing congruences on the instruction sequence model. Two congruences will be used: behavioral congruence and structural congruence.

4 Behavioral and Structural Congruence

X and Y are *behaviorally equivalent* if $[\![X]\!] = [\![Y]\!]$. Behavioral equivalence is not a congruence. For instance $[\![!;!]\!] = S = [\![!;\#0]\!]$ but $[\![\#2;!;!]\!] = S \neq D = [\![\#2;!;\#0]\!]$. This motivates the following definition.

Definition 11.

1. *The set of PGA-contexts is $\mathcal{C} ::= \_ \mid Z;\mathcal{C} \mid \mathcal{C};Z \mid \mathcal{C}^\omega$.*
2. *Let $X, Y \in PGA_\Sigma$. X and Y are* behaviorally congruent *($X =_{\mathrm{bc}} Y$) if for all PGA_Σ-contexts $C[\,]$, $[\![C[X]]\!] = [\![C[Y]]\!]$.*

As a matter of fact it suffices to consider only one kind of context.

Theorem 3. *Let $X, Y \in PGA_\Sigma$. Then*

$$X =_{\mathrm{bc}} Y \Leftrightarrow \forall Z, Z' \in PGA_\Sigma \; [\![Z;X;Z']\!] = [\![Z;Y;Z']\!].$$

Proof. Left to right follows from the definition of behavioral congruence. In order to prove right to left observe first that—because of PGA3—we do not need to consider any contexts of the form $C[\,]^\omega; Z'$ or $Z; C[\,]^\omega; Z'$. The context we do have to consider are therefore the ones given in the table.

1.a	$-$	2.a	$-^\omega$	3.a	$Z''; -^\omega$
1.b	$Z; -$	2.b	$(Z; -)^\omega$	3.b	$Z''; (Z; -)^\omega$
1.c	$-; Z'$	2.c	$(-; Z')^\omega$	3.c	$Z''; (-; Z')^\omega$
1.d	$Z; -; Z'$	2.d	$(Z; -; Z')^\omega$	3.d	$Z''; (Z; -; Z')^\omega$

Assuming the right-hand side, we first show that for every context $C[\ ]$ in the first column we have $[\![C[X]]\!] = [\![C[Y]]\!]$. 1.d is obvious. 1.c follows by taking $Z = \#1$ in 1.d. Now observe that for every U, $[\![U;\#0]\!] = [\![U]\!]$: for finite U this is shown easily with induction to the number of instructions, and for U involving repetition $[\![U;\#0]\!] = [\![U]\!]$ follows from PGA3. This yields 1.a and 1.b by taking $Z' = \#0$ in 1.c. and 1.d, respectively. This covers all contexts in the first column.

We now turn to the third column. We shall first show that for all $n > 0$ and all Z'', $[\![Z'';X^n]\!] = [\![Z'';Y^n]\!]$. The case $n = 1$ has just been established (1.b). Now consider $n+1$: by taking $Z = Z''$ and $Z' = X^n$ in 1.d, $[\![Z'';X;X^n]\!] = [\![Z'';Y;X^n]\!]$. Moreover, from the induction hypothesis it follows that $[\![Z'';Y;X^n]\!] = [\![Z'';Y;Y^n]\!]$. Thus $[\![Z'';X^{n+1}]\!] = [\![Z'';Y^{n+1}]\!]$. From the limit characterization of repetition it now follows that $[\![Z'';X^\omega]\!] = [\![Z'';Y^\omega]\!]$ (3.a). 3.b is dealt with using the same argument with only a small notational overhead. For 3.c and 3.d observe that

$$\begin{aligned}[\![Z'';(X;Z')^\omega]\!] &= [\![Z'';X;(Z';X)^\omega]\!]\\ &= [\![Z'';X;(Z';Y)^\omega]\!]\\ &= [\![Z'';Y;(Z';Y)^\omega]\!]\\ &= [\![Z'';(Y;Z')^\omega]\!]\end{aligned}$$

follows from PGA4, 3.b and 1.d, and

$$\begin{aligned}[\![Z'';(Z;X;Z')^\omega]\!] &= [\![Z'';Z;(X;Z';Z)^\omega]\!]\\ &= [\![Z'';Z;(Y;Z';Z)^\omega]\!]\\ &= [\![Z'';(Z;Y;Z')^\omega]\!]\end{aligned}$$

follows from PGA4 and 3.c. This covers all context in the third column.

Finally we consider the second column. Here every context can be dealt with by taking in the corresponding context in the third column $Z'' = \#1$.

Structural congruence is characterized by the four equation schemes in Table 2. The schemes take care of the simplification of chained jumps. The schemes are termed PGA5-8, respectively. PGA8 can be written as an equation by expanding X, but takes a more 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. Some consequences of these axioms are

$$a;\#2;b;\#0;c = a;\#0;b;\#0;c$$

$$a;\#2;b;\#1;c = a;\#3;b;\#1;c$$

$$a;(\#3;b;c)^\omega = a;(\#0;b;c)^\omega$$

The purpose of structural congruence is to allow successive (and repeating) jumps to be taken together.

Table 2. Equation schemes for structural congruence

$\#n+1;u_1;\ldots;u_n;\#0 = \#0;u_1;\ldots;u_n;\#0$	(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)

Structurally congruent programs are behaviorally congruent as well. This is proven by demonstrating the validity of each closed instance of the structural congruence equations modulo behavioral congruence.

5 The Entry Instruction

As it turns out behavioral congruence on PGA_Σ is not easy to axiomatize by means of equations or conditional equations. It remains an open problem how that can be done. Here the matter will be approached from another angle. First an additional primitive instruction is introduced: @, the entry instruction. The instruction @ in front of a program disallows any jumps into the program otherwise than jumps into the first instruction of the program. Longer jumps are discontinued, and the jump will be carried out as a jump to the control point following @. The entry instruction is new, in the sense that it coincides with no PGA_Σ program or primitive instruction. Its use lies in the fact that it allows an unexpected number of additional (conditional) equations for programs. As a consequence it becomes possible to find a concise final algebra specification of behavioral inequality of programs. This is plausible to some extent: it is much easier to see that programs differ, by finding input leading to different outputs, than to see that they don't differ and hence coincide in the behavioral congruence model of program algebra. The program notation extending PGA_Σ with '@' is denoted $\mathrm{PGA}_{\Sigma,@}$.

In order to provide a mapping from $\mathrm{PGA}_{\Sigma,@}$ into $\mathrm{BPPA}^\infty_\Sigma$ we add to the clauses in Definition 8 the clauses 1.-4. of the following definition

Definition 12.

1. $|@| = D$,
2. $|@;X| = |X|$,
3. $|\#n+1;@| = D$,
4. $|\#n+1;@;X| = |X|$,

and change the clause 2d in Definition 8 into

$$(u \neq @) \Rightarrow |\#k+2;u;X| = |\#k+1;X|.$$

Using these additional rules $[\![\ ]\!]$ can be defined straightforwardly for programs involving the entry instruction. Behavioral congruence has then exactly the same definition in the presence of the entry instruction and Theorem 3 extends trivially to $\mathrm{PGA}_{\Sigma,@}$.

Because programs with different behavior may be considered observationally different it is reasonable to call $\mathrm{PGA}_{\Sigma,@}/{=_{\mathrm{bc}}}$ a fully abstract model. It imposes a maximal congruence under the constraint that observationally different programs will not be identified.

A characterization of behavioral congruence in terms of behavioral equivalence will be given in Theorem 4. The intuition behind this characterization is that behavior extraction abstracts from two aspects that can be recovered by taking into account the influence of a context: the instruction that serves as initial instruction (which for $[\![u_1;\cdots;u_n;\cdots]\!]$ is always u_1) and the difference between divergence and exiting a program with some jump. To make these differences visible at the level of program behaviors only very simple contexts are needed: here are three examples (where $a \neq b$):

$$\#2 \neq_{\mathrm{bc}} \#1 \text{ because } [\![\#2;!;\#0^\omega]\!] = D \neq S = [\![\#1;!;\#0^\omega]\!],$$

$$\#2;a \neq_{\mathrm{bc}} \#2;b \text{ because } [\![\#2;\#2;a]\!] = a \circ D \neq b \circ D = [\![\#2;\#2;b]\!].$$

$$!;\#1 \neq_{\mathrm{bc}} !;\#2 \text{ because } [\![\#2;!;\#1;!;\#0^\omega]\!] = S \neq D = [\![\#2;!;\#2;!;\#0^\omega]\!].$$

Theorem 4. *Let $X, Y \in PGA_{\Sigma,@}$. Then*

1. $X =_{\mathrm{bc}} Y \Leftrightarrow \forall n \in \mathbb{N}\ \forall Z' \in PGA_{\Sigma,@}\ [\![\#n+1;X;Z']\!] = [\![\#n+1;Y;Z']\!]$
2. $X =_{\mathrm{bc}} Y \Leftrightarrow \forall n,m \in \mathbb{N}\ [\![\#n+1;X;!^m;\#0^\omega]\!] = [\![\#n+1;Y;!^m;\#0^\omega]\!]$

Proof. Left to right follows for 1. and 2. from the definition of behavioral congruence.

1. Assume the right-hand side. We employ Theorem 3. Suppose that for some Z, Z', $[\![Z;X;Z']\!] \neq [\![Z;Y;Z']\!]$. Then Z cannot contain an infinite repetition. Therefore it is finite. With induction on the length of Z one then proves the existence of a natural number k such that $[\![\#k+1;X;Z']\!] \neq [\![\#k+1;Y;Z']\!]$. For $l(Z) = (1,0)$ we distinguish 6 cases:
   a) $Z =!$: Then $[\![Z;X;Z']\!] = S = [\![Z;Y;Z']\!]$. Contradiction.
   b) $Z = @$: Then $[\![X;Z']\!] \neq [\![Y;Z']\!]$. Thus also $[\![\#1;X;Z']\!] \neq [\![\#1;Y;Z']\!]$.
   c) $Z = \#n$: As n cannot be 0 we are done.
   d) $Z = a$: Then $a \circ [\![X;Z']\!] \neq a \circ [\![Y;Z']\!]$. Thus $[\![X;Z']\!] \neq [\![Y;Z']\!]$ and hence $[\![\#1;X;Z']\!] \neq [\![\#1;Y;Z']\!]$.
   e) $Z \in \{+a,-a\}$: If $Z = +a$ then

$$[\![X;Z']\!] \trianglelefteq a \trianglerighteq [\![\#2;X;Z']\!] \neq [\![Y;Z']\!] \trianglelefteq a \trianglerighteq [\![\#2;Y;Z']\!].$$

   Then $[\![X;Z']\!] \neq [\![Y;Z']\!]$ or $[\![\#2;X;Z']\!] \neq [\![\#2;Y;Z']\!]$. In the latter case we are done and in the first case we can take $k = 0$. $-a$ is dealt with similarly.

Now consider $l(Z) = (m+2, 0)$. We have to distinguish 10 cases. Seven cases correspond to the repetition-free clauses in 2 of Definition 8. They follow from a straightforward appeal to the induction hypothesis. The remaining three cases correspond to 2.–4. of Definition 12.

a) $Z = @; Z''$: Then $[\![Z''; X; Z']\!] \neq [\![Z''; Y; Z']\!]$. Hence $[\![\#k+1; X; Z']\!] \neq [\![\#k+1; Y; Z']\!]$ for some k by the induction hypothesis.
b) $Z = \#n+1; @$: Then $[\![X; Z']\!] \neq [\![Y; Z']\!]$. Hence $[\![\#1; X; Z']\!] \neq [\![\#1; Y; Z]\!]$.
c) $Z = \#n+1; @; Z''$: Then $[\![Z''; X; Z']\!] \neq [\![Z''; Y; Z']\!]$ and we can again apply the induction hypothesis.

2. Assume the right-hand side. We make an appeal to 1. Suppose there are k and Z' such that $[\![\#k+1; X; Z']\!] \neq [\![\#k+1; Y; Z']\!]$. If both X and Y are infinite then $[\![\#k+1; X]\!] \neq [\![\#k+1; Y]\!]$ and hence also $[\![\#k+1; X; \#0^\omega]\!] \neq [\![\#k+1; Y; \#0^\omega]\!]$. Suppose only one of the two, say Y, has a repetition, then writing $X = u_1; \ldots; u_n$, it follows that: $[\![\#k+1; u_1; \ldots; u_n; Z']\!] \neq [\![\#k+1; Y]\!]$. At this point an induction on n can be used to establish the existence of an m with $[\![\#k+1; u_1; \ldots; u_n; !^m; \#0^\omega]\!] \neq [\![\#k+1; Y]\!]$ and hence $[\![\#k+1; u_1; \ldots; u_n; !^m; \#0^\omega]\!] \neq [\![\#k+1; Y; !^m; \#0^\omega]\!]$. If both X and Y are finite instruction sequences, an induction on their maximum length suffices to obtain the required fact (again involving a significant case ramification).

Example 1.

1. $@; ! =_{\mathrm{bc}} !^\omega$ since for all n, Z, $[\![\#n+1; @; !; Z]\!] = [\![!; Z]\!] = S = [\![\#n+1; !^\omega; Z]\!]$, and
2. $@; \#0 =_{\mathrm{bc}} \#0^\omega$ since for all n, Z, $[\![\#n+1; @; \#0; Z]\!] = [\![\#0; Z]\!] = D = [\![\#n+1; \#0^\omega; Z]\!]$.

The characterization above suggests that behavioral congruence may be undecidable. This of course is not the case: the quantifier over m can be bounded because m need not exceed the maximum of the counters of jump instructions in X and Y plus 1. An upper bound for n is as follows: if $l(X) = (k, m)$ and $l(Y) = (k', m')$ then $(k+m) \times (k'+m')$ is an upper bound of the n's that must be checked.

Programs starting with the entry instruction can be distinguished by means of simpler contexts:

Corollary 1. *Let $X, Y \in PGA_{\Sigma,@}$. Then*

1. $@; X =_{\mathrm{bc}} @; Y \Leftrightarrow \forall n \in \mathbb{N} [\![X; !^n; \#0^\omega]\!] = [\![Y; !^n; \#0^\omega]\!]$
2. $@; X =_{\mathrm{bc}} @; Y \Leftrightarrow \forall Z [\![X; Z]\!] = [\![Y; Z]\!]$

Proof. 1. and 2. follow from that fact that for every $n, k \in \mathbb{N}$ and every X, $[\![\#k+1; @; X; !^n; \#0^\omega]\!] = [\![X; !^n; \#0^\omega]\!]$ and $[\![\#k+1; @; X; Z]\!] = [\![X; Z]\!]$.

Since $[\![X]\!] = [\![X; \#0^\omega; Z]\!]$ for all program expressions X and Z, it follows from Corollary 1.2 that behavioral equivalence can be recovered from behavioral congruence in the following way:

Corollary 2. *Let $X, Y \in PGA_{\Sigma,@}$. Then*

$$X =_{\mathrm{be}} Y \Leftrightarrow @; X; \#0^\omega =_{\mathrm{bc}} @; Y; \#0^\omega.$$

Programs ending with an entry instruction allow a simpler characterisation as well:

Corollary 3. *Let $X, Y \in PGA_{\Sigma,@}$. Then $X; @ =_{\mathrm{bc}} Y; @$ iff for all $n \in \mathbb{N}$,*

$$[\![\#n+1; X; !^\omega]\!] = [\![\#n+1; Y; !^\omega]\!] \;\&\; [\![\#n+1; X; \#0^\omega]\!] = [\![\#n+1; Y; \#0^\omega]\!]$$

Proof. '$\Rightarrow$': Suppose that $X; @ =_{\mathrm{bc}} Y; @$, then for all n and m,

$$(\star) \quad [\![\#n+1; X; @; !^m; \#0^\omega]\!] = [\![\#n+1; Y; @; !^m; \#0^\omega]\!].$$

Then

$$\begin{aligned}
[\![\#n+1; X; !^\omega]\!] &= [\![\#n+1; X; !^\omega; \#0^\omega]\!] \\
&= [\![\#n+1; X; @; !; \#0^\omega]\!] \text{ since } @; ! =_{\mathrm{bc}} !^\omega \text{ (Example 1)} \\
&= [\![\#n+1; Y; @; !; \#0^\omega]\!] \text{ take in } (\star)\ m = 1 \\
&= [\![\#n+1; Y; !^\omega; \#0^\omega]\!] \\
&= [\![\#n+1; Y; !^\omega]\!]
\end{aligned}$$

Similarly

$$\begin{aligned}
[\![\#n+1; X; \#0^\omega]\!] &= [\![\#n+1; X; \#0^\omega; \#0^\omega]\!] \\
&= [\![\#n+1; X; @; \#0; \#0^\omega]\!] \text{ since } @; \#0 =_{\mathrm{bc}} \#0^\omega \text{ (Example 1)} \\
&= [\![\#n+1; X; @; \#0^\omega]\!] \\
&= [\![\#n+1; Y; @; \#0^\omega]\!] \qquad \text{take in } (\star)\ m = 0 \\
&= [\![\#n+1; Y; @; \#0; \#0^\omega]\!] \\
&= [\![\#n+1; Y; \#0^\omega; \#0^\omega]\!] \\
&= [\![\#n+1; Y; \#0^\omega]\!]
\end{aligned}$$

'$\Leftarrow$': for $m = 0$, the above argument runs in the other direction

$$\begin{aligned}
[\![\#n+1; X; @; !^0; \#0^\omega]\!] &= [\![\#n+1; X; @; \#0^\omega]\!] \\
&= [\![\#n+1; X; @; \#0; \#0^\omega]\!] \\
&= [\![\#n+1; X; \#0^\omega; \#0^\omega]\!] \\
&= [\![\#n+1; Y; \#0^\omega; \#0^\omega]\!] \\
&= [\![\#n+1; Y; @; \#0; \#0^\omega]\!] \\
&= [\![\#n+1; Y; @; \#0^\omega]\!] \\
&= [\![\#n+1; Y; @; !^0; \#0^\omega]\!]
\end{aligned}$$

The case $m > 0$ is similar.

6 Axiomatization of the Fully Abstract Model

With $\mathrm{CEQ}_@$ the collection of 20 equations and inequations in Table 3 will be denoted (CEQ for, 'conditional and unconditional equations'). They can be viewed

Table 3. $\mathrm{CEQ}_{@}$

(1)	$@;! = !^{\omega}$
(2)	$@;\#0 = \#0^{\omega}$
(3)	$@;@ = @$
(4)	$\#n+1;@ = @$
(5)	$+a;@ = a;@$
(6)	$-a;@ = a;@$
(7)	$\#n+l+1;u_1;\ldots;u_n;@ = \#n+1;u_1;\ldots;u_n;@$
(8)	$@;u_1;\ldots;u_n;@ = @;u_1;\ldots;u_n;\#1 \quad (\forall 1 \leq j \leq n\ u_j = \#k \Rightarrow k+j \leq n+1)$
(9)	$@;u_1;\ldots;u_n;@ = @;u_1;\ldots;u_n;\#1 \Rightarrow @;(u_1;\ldots;u_n;@)^{\omega} = @;(u_1;\ldots;u_n;\#1)^{\omega}$
(10)	$@;\#1 = @$
(11)	$@;\#n+2;u = @;\#n+1 \quad (\text{if } u \neq @)$
(12)	$@;a;@ = @;a$
(13)	$@;a = @;+a;\#1$
(14)	$@;-a = @;+a;\#2$
(15)	$@;X = @;Y\ \&\ @;\#2;X = @;\#2;Y \Leftrightarrow @;+a;X = @;+a;Y$
(16)	$@;u;X = @;v;X \Rightarrow u;X = v;X$
(17)	$@;! \neq @;\#j$
(18)	$@;! \neq @;+a;X$
(19)	$@;\#0 \neq @;+a;X$
(20)	$@;+a;X \neq @;+b;Y \quad (a \neq b \in \Sigma)$

as axioms from which other facts may be derived using conditional equational logic. Inequations can be understood as a shorthand for conditional equation: e.g. $@;! = @;\#j \Rightarrow X = Y$ represents $@;! \neq @;\#j$. No attempt has been made to minimize or optimize this collection. We shall first show that $\mathrm{CEQ}_{@}$ is valid in $\mathrm{PGA}_{\Sigma,@}/=_{\mathrm{bc}}$.

Proposition 4. *$PGA_{\Sigma,@}/=_{\mathrm{bc}} \models CEQ_{@}$*

Proof. 1. See Example 1.1.

2. See Example 1.2.
3. Since $[\![@;@;Z]\!] = [\![@;Z]\!]$ for all Z, we can apply Corollary 1.2.
4. If $k = 0$, $[\![\#k+1;\#n+1;@;Z]\!] = [\![\#1;\#n+1;@;Z]\!] = [\![\#n+1;@;Z]\!] = [\![@;Z]\!] = [\![\#k+1;@;Z]\!]$ and if $k > 0$ $[\![\#k+1;\#n+1;@;Z]\!] = [\![\#k;@;Z]\!] = [\![@;Z]\!] = [\![\#k+1;@;Z]\!]$. Now apply Theorem 4.1.
5. We apply again Theorem 4.1. For $k > 0$ the process extraction equations match both sides. For $k = 0$ we obtain: $[\![\#1;+a;@;Z]\!] = [\![+a;@;Z]\!] = [\![@;Z]\!] \trianglelefteq a \trianglerighteq [\![\#2;@;Z]\!] = [\![@;Z]\!] \trianglelefteq a \trianglerighteq [\![@;Z]\!] = a \circ [\![@;Z]\!] = [\![a;@;Z]\!] = [\![\#1;a;@;Z]\!]$. For $k > 0$ we have

$$[\![\#k+1;+a;@;Z]\!] = [\![\#k;@;Z]\!] = [\![\#k+1;a;@;Z]\!].$$

6. Similar to 5.
7. For $n = 1$, $[\![\#k+2;u_1;@]\!] = [\![\#k+1;@]\!] = [\![\#1;@]\!] = [\![\#2;u_1;@]\!]$ if $u_1 \neq @$, and otherwise $[\![\#k+2;@;@]\!] = [\![@]\!] = [\![\#2;@;@]\!]$. For $n > 1$ we apply the induction hypothesis.

8. This follows from the fact that the entry instruction simply behaves as a skip if it does not affect preceding jumps; that is, if the jumps are small enough to be not affected by discontinuation.
9. Let $\boldsymbol{u} = u_1; \ldots; u_n$ and suppose $@; \boldsymbol{u}; @ =_{\mathrm{bc}} @; \boldsymbol{u}; \#1$. We shall show by induction on l that

$$@; (\boldsymbol{u}; @)^l =_{\mathrm{bc}} @; (\boldsymbol{u}; \#1)^l$$

for all $l > 0$. The base case follows from the assumption. For $l + 2$ we have

$$\begin{aligned}
[\![(\boldsymbol{u}; @)^{l+2}; Z]\!] &= [\![(\boldsymbol{u}; @)^l; \boldsymbol{u}; @; \boldsymbol{u}; @; Z]\!] \\
&= [\![(\boldsymbol{u}; @)^l; \boldsymbol{u}; @; \boldsymbol{u}; \#1; Z]\!] \text{ by the assumption} \\
&= [\![(\boldsymbol{u}; @)^{l+1}; \boldsymbol{u}; \#1; Z]\!] \\
&= [\![(\boldsymbol{u}; \#1)^{l+1}; \boldsymbol{u}; \#1; Z]\!] \quad \text{by the induction hypothesis} \\
&= [\![(\boldsymbol{u}; \#1)^{l+2}; Z]\!]
\end{aligned}$$

Thus also $@; (\boldsymbol{u}; @)^{l+2} =_{\mathrm{bc}} @; (\boldsymbol{u}; \#1)^{l+2}$ by Corollary 1.2 and hence $[\![(\boldsymbol{u}; @)^l]\!] = [\![@; (\boldsymbol{u}; @)^l]\!] = [\![@; (\boldsymbol{u}; \#1)^l]\!] = [\![(\boldsymbol{u}; \#1)^l]\!]$ for all $l > 0$. It follows that $[\![(\boldsymbol{u}; @)^\omega]\!] = [\![(\boldsymbol{u}; \#1)^\omega]\!]$. Therefore we have $[\![(\boldsymbol{u}; @)^\omega; Z]\!] = [\![(\boldsymbol{u}; \#1)^\omega; Z]\!]$ for all Z. Thus $@; (\boldsymbol{u}; @)^\omega =_{\mathrm{bc}} @; (\boldsymbol{u}; \#1)^\omega$ by Corollary 1.2.
10. Since $[\![\#1; @; Z]\!] = [\![@; Z]\!] = [\![Z]\!]$ for all Z, we can apply Corollary 1.2.
11. By Corollary 1.2 since for all Z, $[\![\#n + 2; u; Z]\!] = [\![\#n + 1; Z]\!]$ if $u \neq @$.
12. Again by Corollary 1.2 since for all Z, $[\![a; @; Z]\!] = a \circ [\![Z]\!] = [\![a; Z]\!]$.
13. Similar to (12).
14. Similar to (13).
15. This follows straightforwardly from Corollary 1.2 and the fact that

$$\forall Z [\![X; Z]\!] = [\![Y; Z]\!] \;\&\; [\![\#2; X; Z]\!] = [\![\#2; Y; Z]\!]$$

iff

$$\forall Z [\![X; Z]\!] \trianglelefteq a \trianglerighteq [\![\#2; X; Z]\!] = [\![Y; Z]\!] \trianglelefteq a \trianglerighteq [\![\#2; Y; Z]\!].$$

16. Apply Theorem 4.1.
17. Since $[\![@; !]\!] = S \neq D = [\![@; \#j]\!]$.
18. Since $[\![@; !]\!] = S \neq [\![X]\!] \trianglelefteq a \trianglerighteq [\![\#2; X]\!] = [\![@; +a; X]\!]$.
19. Since $[\![@; \#0]\!] = D \neq [\![X]\!] \trianglelefteq a \trianglerighteq [\![\#2; X]\!] = [\![@; +a; X]\!]$.
20. Since $[\![@; +a; X]\!] = [\![X]\!] \trianglelefteq a \trianglerighteq [\![\#2; X]\!] \neq [\![Y]\!] \trianglelefteq b \trianglerighteq [\![\#2; Y]\!] = [\![@; +b; Y]\!]$.

The axiom system PGA1-8 + $\mathrm{CEQ}_@$ is obtained by combining the equations for instruction sequence congruence, the axioms for structural equivalence and the axioms of $\mathrm{CEQ}_@$. From the previous proposition it follows that this system is sound, i.e. applying its axioms and the rules of conditional equational logic always yields equations that are valid in $\mathrm{PGA}_{\Sigma,@}/ =_{\mathrm{bc}}$. The converse, i.e. provable equality of behavioral congruence, can be shown in the repetition-free case. Completeness for infinite programs remains an open problem.

Theorem 5. *PGA1-8 + $CEQ_@$ is complete for finite programs, i.e. for repetition-free $X, Y \in PGA_{\Sigma,@}$,*

$$X =_{\mathrm{bc}} Y \Leftrightarrow PGA1\text{-}8 + CEQ_@ \vdash X = Y$$

Proof. Right to left follows from the previous proposition. To prove the other direction, first notice that in the absence of entry instructions lengths must be equal, or else a separating context can be easily manufactured. Then, still without @, the fact is demonstrated with induction to program lengths, using (16) as a main tool, in addition to a substantial case distinction.

In the presence of entry instructions, (7) and (8) are used to transform both programs to instruction sequences involving at most a single entry instruction. If only one of the programs contains an entry instruction a separating context is found using a jump that can jump over the program without entry instruction entirely while halting at the other program's entry instruction. At this point it can be assumed that $X = X_1; @; X_2$ and $Y = Y_1; @; Y_2$. Let k be the maximum of the lengths of X_1 and Y_1, then $[\![\#k+1; X_1; @; X_2]\!] = [\![@; X_2]\!]$ and $[\![\#k+1; Y_1; @; Y_2]\!] = [\![@; Y_2]\!]$. Now $@; X_2$ and $@; Y_2$ can be proven equal, and this is shown by means of an induction on the sum of the lengths of both. Finally the argument is concluded by an induction on the sum of the lengths of X_1 and Y_1.

7 A Final Algebra Specification for Behavioral Congruence

In this section we shall show that PGA1-8 + $CEQ_@$ constitutes a final algebra specification of the fully abstract program algebra with entry instruction.

Lemma 2. *Let $X \in PGA_{\Sigma,@}$. Then*

1. $[\![X]\!] = S \Rightarrow PGA1\text{-}8 + CEQ_@ \vdash @; X = @; !$
2. $[\![X]\!] = D \Rightarrow PGA1\text{-}8 + CEQ_@ \vdash @; X; \#0^\omega = @; \#0$
3. $[\![X]\!] = P \trianglelefteq a \trianglerighteq Q \Rightarrow PGA1\text{-}8 + CEQ_@ \vdash @; X = @; +a; Y$ *for some* $Y \in PGA_{\Sigma,@}$

Proof. We shall write $\vdash$ instead of PGA1-8 + $CEQ_@ \vdash$ and consider the definition of $|X|$ as a collection of rewrite rules, working modulo instruction sequence equivalence (for which PGA1-4 are complete).

1. The assumption implies that after finitely many rewrites the result S is obtained. We use induction on the length of this rewrite sequence. If one step is needed (the theoretical minimum), there are two cases: $X =!$, or $X =!; Y$ for some Y. The first case is immediate; the second case follows by $\vdash @; X = @; !; Y =!^\omega; Y =!^\omega = @; !$ employing (1). If $k+1$ steps are needed the last step must be either a rewrite of a jump or the removal of an entry instruction. We only consider the first case. Thus $X = \#n; Y$ for some Y. If $n = 1$ then $|Y| = S$ and hence $\vdash @; Y = @; !$ by the induction hypothesis.

Thus $\vdash @;X = @;\#1;Y = @;Y = @;!$ by (10). If $X = \#n+2;u;Y$ there are two cases: u is the entry instruction, or not. Assume that it is not. Then $|\#n+1;Y| = S$. Using the induction hypothesis and (11) it follows that $\vdash @;X = @;\#n+2;u;Y = @;\#n+1;Y = @;!$. If u is the entry instruction we have $\vdash @;X = @;\#n+2;@;Y = @;@;Y = @;Y = @;!$ by (3), (4) and the induction hypothesis.

2. A proof of this fact uses a case distinction: either in finitely many steps the rewriting process of the process extraction leads to $\#0;Z$ for some Z, or an infinite sequence of rewrites results which must be of a cyclic nature. In the fist case induction on the number of rewrite steps involved provides the required result without difficulty. The structural congruence equations will not be needed in this case. In the case of an infinite rewrite it follows that the rewriting contains a circularity. By means of the chaining of successive jumps the expression can be rewritten into an expression in which a single jump, contained in the repeating part traverses the whole repeating part and then chains with itself. PGA7 can be used to introduce an instruction $\#0$, thereby reducing the case to the previous one. This is best illustrated by means of an example.

$$
\begin{aligned}
& @;\#5;!;\#0;(\#4;+a;\#2;!;\#1)^\omega \\
= {} & @;\#5;!;\#0;(\#5;+a;\#2;!;\#1)^\omega && \text{PGA6} \\
= {} & @;\#5;!;\#0;(\#0;+a;\#2;!;\#1)^\omega && \text{PGA7} \\
= {} & @;\#5;!;\#0;\#0;+a;\#2;!;\#1;(\#0;+a;\#2;!;\#1)^\omega && \text{PGA4} \\
= {} & @;\#5;!;\#1;\#0;+a;\#2;!;\#1;(\#0;+a;\#2;!;\#1)^\omega && \text{PGA5} \\
= {} & @;\#2;!;\#1;(\#0;+a;\#2;!;\#1)^\omega && \text{PGA4} \\
= {} & @;\#1;(\#0;+a;\#2;!;\#1)^\omega && (11) \\
= {} & @;(\#0;+a;\#2;!;\#1)^\omega && (10) \\
= {} & @;\#0;+a;\#2;!;\#1;(\#0;+a;\#2;!;\#1)^\omega && \text{PGA4} \\
= {} & \#0^\omega;+a;\#2;!;\#1;(\#0;+a;\#2;!;\#1)^\omega && (2) \\
= {} & \#0^\omega && \text{PGA3} \\
= {} & @;\#0 && (2).
\end{aligned}
$$

3. This fact follows by means of an induction on the number of rewrite steps needed for the program extraction operator to arrive at an expression of the form $P \trianglelefteq a \trianglerighteq Q$.

The results can be taken together in the following theorem which can be read as follows: 'PGA_{1-8}+$\mathrm{CEQ}_@$ constitutes a final algebra specification of the fully abstract program algebra with entry instruction'.

Proposition 5.

$$[\![X]\!] \neq [\![Y]\!] \Rightarrow \mathrm{PGA}_{1-8} + \mathrm{CEQ}_@ \vdash @;X \neq @;Y.$$

Proof. With induction on n it will beshown that $\pi_n([\![X]\!]) \neq \pi_n([\![Y]\!])$ implies the provability of $@;X \neq @;Y$. The basis is immediate because zero'th projections are D in both cases, and a diference cannot exist. Then suppose that

$\pi_{n+1}([\![X]\!]) \neq \pi_{n+1}([\![Y]\!])$ A case distinction has to be analysed. Suppose $[\![X]\!] = S$ and $[\![Y]\!] = D$. Then $\mathrm{PGA}_{1-8} + \mathrm{CEQ}_{@}, \vdash @; X = @; !$ and $\mathrm{PGA}_{1-8} + \mathrm{CEQ}_{@}, \vdash @; X = @; \#0$ by the previous lemma. Thus $\mathrm{PGA}_{1-8} + \mathrm{CEQ}_{@} \vdash @; X \neq @; Y$ using (17). All other cases are similar except one: $[\![X]\!] = P \trianglelefteq a \trianglerighteq Q$ and $[\![Y]\!] = P' \trianglelefteq a \trianglerighteq Q'$. Then there must be X' and Y' such that $\mathrm{PGA}_{1-8} + \mathrm{CEQ}_{@}, \vdash @; X = @; +a; X'$ and $\mathrm{PGA}_{1-8} + \mathrm{CEQ}_{@}, \vdash @; Y = @; +a; Y'$. It then follows that either $\pi_n([\![X']\!]) \neq \pi_n([\![Y']\!])$ or $\pi_n([\![\#2; X']\!]) \neq \pi_n([\![\#2; Y']\!])$. In both cases the induction hypothesis can be applied. Finally (15) is applied to obtain the required fact.

Theorem 6.

$$X \neq_{bc} Y \Rightarrow \mathrm{PGA}_{1-8} + \mathrm{CEQ}_{@} \vdash X \neq Y.$$

Proof. If $X \neq_{bc} Y$ then for some P and Q, $[\![P; X; Q]\!] \neq [\![P; Y; Q]\!]$. Using the previous proposition $\mathrm{PGA}_{1-8} + \mathrm{CEQ}_{@} \vdash @; P; X; Q \neq @; P; Y; Q$. This implies $\mathrm{PGA}_{1-8} + \mathrm{CEQ}_{@} \vdash X \neq Y$ by the laws of conditional equational logic.

8 Concluding Remarks

Polarized process algebra has been used in order to give a natural semantics for programs.

The question how to give an equational initial algebra specification of the program algebra (with or without entry instruction) modulo behavioral congruence remains open. As stated in [3] behavioral congruence is decidable on PGA expressions. For that reason an infinite equational specification exists. The problem remains to present such a specification either with a finite set of equations or with the help of a few comprehensible axiom schemes. General specification theory (see [4]) states that a finite equational specification can be found which is an orthogonal rewrite system (see [9,5]) at the same time, probably at the cost of some auxiliary functions. Following the proof strategy of [4] an unreadable specification will be obtained, however. The problem remains to obtain a workable specification with these virtues. Thus as it stands both finding an initial algebra specification and finding a 'better' final algebra specification (only finitely many equations, no additional object) for program algebra with behavioral congruence are open matters.

Another question left open for further investigation is whether the entry instruction can be naturally combined with the unit instruction operator as studied in [10]. This seems not to be the case. A similar question can be posed regarding the repetition instruction mentioned in [3].

References

1. J.A. Bergstra and J.-W. Klop. Process algebra for synchronous communication. *Information and Control*, 60 (1/3):109–137, 1984.

2. J.A. Bergstra and M.E. Loots. Program algebra for component code. *Formal Aspects of Computing*, 12(1):1–17, 2000.
3. J.A. Bergstra and M.E. Loots. Program algebra for sequential code. *Journal of Logic and Algebraic Programming*, 51(2):125–156, 2002.
4. J.A. Bergstra and J.V. Tucker. Equational specifications, complete rewriting systems and computable and semi-computable algebras. *Journal of the ACM*, 42(6):1194–1230, 1995.
5. I. Bethke. Completion of equational specifications. In Terese, editors, *Term Rewriting Systems*, Cambridge Tracts in Theoretical Computer Science 55, pages 260–300, Cambridge University Press, 2003.
6. S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(8):560–599, 1984.
7. J.W. de Bakker and J.I. Zucker. Processes and the denotational semantics of concurreny. *Information and Control*, 54(1/2):70–120, 1982.
8. W.J. Fokkink. Axiomatizations for the perpetual loop in process algebra. In P. Degano, R. Gorrieri, and A. Machetti-Spaccamela, editors, *Proceedings of the 24-th ICALP, ICALP'97*, Lecture Notes in Comp. Sci. 1256, pages 571–581. Springer Berlin, 1997.
9. J.-W. Klop. Term rewriting systems. In *Handbook of Logic in Computer Science*, volume II, pages 1–116. Oxford University Press, 1992.
10. A. Ponse. Program algebra with unit instruction operators. *Journal of Logic and Algebraic Programming*, 51(2):157–174, 2002.
11. V. Stoltenberg-Hansen, I. Lindström, and E.R. Griffor. *Mathematical Theory of Domains*, Cambridge Tracts in Theoretical Computer Science 22, Cambridge University Press, 1994.