Projection Semantics for Rigid Loops

Jan A. Bergstra [a,b] Alban Ponse [a]

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

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

Abstract

A rigid loop is a for-loop with a counter not accessible to the loop body or any other part of a program. Special instructions for rigid loops are introduced on top of the syntax of the program algebra PGA. Two different semantic projections are provided and proven equivalent. One of these is taken to have definitional status on the basis of two criteria: 'normative semantic adequacy' and 'indicative algorithmic adequacy'.

Key words: Program algebra, For-loop, Projection semantics.

1 Introduction

In this paper we extend the program algebra PGA [5] with several new instructions to deal with so-called rigid loops. Rigid loops are just program fragments that impose the repetition of a body a fixed number of times. Rigid loops are very limited in expressive power. Indeed finite state PGA-programs with rigid loops can be projected into equivalent finite state PGA-programs without rigid loops at the cost of a combinatorial explosion in length. Like non-recursive procedures, rigid loops may be of use when investigating options for compiler writing for specific processor architectures. Our specific motivation to consider rigid loops arose when studying the potential gains that may arise from microthread multiplexing on a single pipelined instruction processing architecture. Following Jesshope *et al.* in [11,13] loops and nested loops can be usefully split into microthreads which then may be scheduled either in a multiplexed fashion on a single pipeline, in an attempt to make use of the unavoidable clock cycles in which a single thread on the pipeline features stalling, or concurrently on parallel pipelines on a multiple pipeline architecture in order to maximize the processing speed for an originally sequential program. Rigid loops have the advantage of simplifying dependency analysis, thus shedding more easily light on what one might hope to achieve. As it turns out rigid loops are quite interesting even without applications like the one just mentioned in mind as a case study for projection semantics. Projection semantics has been advocated in [5] as a formal modeling technique close to programmers intuitions. Rigid loops can be easily provided with a projection semantics at the cost of a combinatorial explosion in program length. Here it will be argued that this is not the most appropriate way to deal with this issue and another style of projection which avoids this drastic blow-up in program length is provided and proven semantically equivalent but algorithmically more natural.

Although [5] provides a clear statement on the objectives and merits of projection semantics, it fails to provide a methodology which scales to full size program notations by its exclusive focus on semantic issues. Projection semantics provides the meaning of a program notation, say PGLX, by means of a mapping `pglx2pga` from PGLX to PGA which assigns to each entity in PGLX a program object (i.e., an element of a program algebra, in this case PGA). The program objects used are finite or infinite instruction streams, over a limited set of primitive instructions which goes with the program algebra. As a semantic strategy projection semantics is independent of this particular program algebra, but we will use PGA because it works and it allows for a very slow build up of features, thus permitting a very gradual growth in expressiveness. The key dogma of projection semantics is that an entity is a program by either being or representing an instruction stream. Instruction streams are program objects, i.e., mathematical entities that stand for programs. Thus a projection explains how some entity can be considered an instruction stream and only by explaining (by way of a projection) what instruction stream an entity represents it can be considered a program. It is more precise always to speak of a program representation rather than of a program but because that is very uncommon the term ‘program’ is used also in cases that a projection does not speak for itself.

Until a projection has been fixed for an entity it is is a candidate program rather than a program. Only by fixing its projection into an instruction stream a candidate program becomes a program, comparable to how a document becomes legally binding by the addition of relevant signatures, locations and dates. We do not accept the conventional viewpoint that a program can be given a new meaning. Rather a candidate program can be made to stand for another program by changing its projection, just as a contract changes when one modifies the signatures. Candidate programs may have a quite convincing syntax suggesting meaning without further ado. We believe that this is *never* actually true. The operational meaning of candidate programs always requires detailed description covering a variety of circumstances. Now ‘projection semantics’ as a style of providing programming language semantics will have to deal with many notations that are already in practical use and that may have the status of candidate programs from the perspective of program algebra based projection semantics, but for which quite satisfactory semantic descriptions have been found by means of other techniques. Here we are dealing with providing projection semantics for 'known' program notations and the question may arise as to which semantic description technique is most effective.

Claiming definitional status for a projection for a program notation that has been given a semantic description already is clearly problematic. Therefore the claim can go no further than that a projection might be considered to have normative strength semantically, under the hypothesis that it would be the only description at hand, accepting that in many cases it will have not have definitional status simply because other definitions have that status already. Such a projection, for a known and well specified program notation will be called a reconstruction projection semantics in order to acknowledge that a definitional status is not claimed. This leads to the position that for Pascal one may achieve no more than a reconstruction projection semantics while for Perl a projection semantics might still be achievable.

For new or unknown notations, however, whether useful or not, a projection can be claimed to contain primary semantic information which by definition cannot be validated or verified against any other description, because of its normative nature. Of course validation is possible: by means of a projection semantics an operational meaning is assigned to syntactic constructs (assuming a string based source language) in a candidate program notation. Because the syntax of this candidate program notation is itself a matter of meticulous design the operational meaning should make best possible use of the syntax that has been made available. If a projection prescribes an unintelligible meaning to a construct that might have been given a clear and useful meaning instead a design error has occurred which can and probably should be repaired.

Returning to the issue that known program notations cannot be given a projection semantics the following solution to this somewhat philosophical issue can be found. For projection semantics as a topic of investigation this philosophical matter is simply solved by always using slightly unconventional syntax (however marginal the differences) such that the setting establishes a new syntax which is given a meaning for the first and therefore definitive time. The ability of a projection for a candidate program notation to serve as a carrier of intended semantic information is termed *normative semantic adequacy*. Normative semantic adequacy does not come for free: it requires that comprehensible projections into comprehensible programs are used to provide a realistic, suggestive and useful meaning (in terms of instruction streams) for new syntax. Usually a projection will be into a program notation that has been provided with a projection semantics already thus giving rise to chains of projections.

Besides normative semantic adequacy one also expects a projection to represent an indication (or model) of how the actual processing of a (candidate) program might in practice proceed. Exponential or even polynomial blow-up of the size of an entity during its projecting transformation are signs that *indicative algorithmic adequacy* has not been achieved.

A projection for a programming notation feature which enjoys both normative semantic adequacy and indicative algorithmic adequacy is called a defining projection. If it uses some services of type T it will be called a T service based defining projection. Using this terminology we will develop in this paper a rigid loop counter service based defining projection for PGArl (PGA with rigid loops).

The further content of this paper is divided into four parts: in Section 2 we formally introduce threads and services. In Section 3 we introduce the program algebra PGA, thread extraction and a further extension of PGA. In Section 4 we extend PGA with rigid loops to PGArl, including two forms of projection semantics. It is clarified that the projection semantics making use of decreasing loop counters enjoys both normative semantic adequacy and indicative algorithmic adequacy and that the pure projection into PGA fails for the second criterion. The paper is ended with some conclusions in Section 5.

2 Threads and Services

The behavior of programs under execution is modelled by *threads*. In this section we introduce thread algebra. Then we introduce services, devices that can be *used* by a thread in order to increase expressiveness.

2.1 Thread algebra

Basic thread algebra, or BTA for short, is intended for the description of sequential program behavior (see [6]; in [5] BTA is introduced as *basic polarized process algebra*). Based on a finite set A of *actions* it has the following constants and operators:

- the *termination* constant S,
- the *deadlock* or *inaction* constant D,
- for each $a \in A$, a binary *postconditional composition* operator $\_ \trianglelefteq a \trianglerighteq \_$.

We use *action prefixing* $a \circ P$ as an abbreviation for $P \trianglelefteq a \trianglerighteq P$ and take $\circ$ to bind strongest. Furthermore, for $n \in \mathbb{N}$ we define $a^n \circ P$ by $a^0 \circ P = P$ and $a^{n+1} \circ P = a \circ (a^n \circ P)$.

The operational intuition behind thread algebra is that each action represents a request to be processed by the execution environment. At completion of the processing of the request, the environment produces a reply value true or false to the thread under execution and may undergo a change of state. The thread $P \trianglelefteq a \trianglerighteq Q$ will then proceed as P if the processing of a yielded the reply true indicating successful processing, and it will proceed as Q if the processing of a yielded the reply false.

BTA can be equipped with a partial order and an *approximation operator*.

(1) $\sqsubseteq$ is the partial ordering on BTA generated by the clauses
  (a) for all $P \in$ BTA, $\mathsf{D} \sqsubseteq P$, and
  (b) for all $P_1, P_2, Q_1, Q_2 \in$ BTA, $a \in A$,

$$P_1 \sqsubseteq Q_1 \ \&\ P_2 \sqsubseteq Q_2 \Rightarrow P_1 \trianglelefteq a \trianglerighteq P_2 \sqsubseteq Q_1 \trianglelefteq a \trianglerighteq Q_2.$$

(2) $\pi : \mathbb{N} \times \mathrm{BTA} \to \mathrm{BTA}$ is the approximation operator determined by the equations
  (a) for all $P \in$ BTA, $\pi(0, P) = \mathsf{D}$,
  (b) for all $n \in \mathbb{N}$, $\pi(n+1, \mathsf{S}) = \mathsf{S}$, $\pi(n+1, \mathsf{D}) = \mathsf{D}$, and
  (c) for all $P, Q \in \mathrm{BTA}, n \in \mathbb{N}$,

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

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

The operator π finitely approximates every thread in BTA. That is, for all $P \in$ BTA,

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

Threads can be finite or infinite. Following the metric theory of [1] as the basis of processes in [4], BTA has a completion BTA^∞ which comprises also infinite threads. Standard properties of the completion technique yield that we may take BTA^∞ as the cpo consisting of all so-called *projective* sequences. That is,

$$\mathrm{BTA}^\infty = \{(P_n)_{n\in\mathbb{N}} \mid \forall n \in \mathbb{N}\ (P_n \in \mathrm{BTA}\ \&\ \pi_n(P_{n+1}) = P_n)\}$$

with

$$(P_n)_{n\in\mathbb{N}} \sqsubseteq (Q_n)_{n\in\mathbb{N}} \Leftrightarrow \forall n \in \mathbb{N}\ P_n \sqsubseteq Q_n$$

and

$$(P_n)_{n\in\mathbb{N}} = (Q_n)_{n\in\mathbb{N}} \Leftrightarrow \forall n \in \mathbb{N}\ P_n = Q_n.$$

(For a detailed account of this construction see [3].)

Let $I = \{1, ..., n\}$ for some $n > 0$. A *finite linear recursive specification* over BTA is a set of equations

$$X_i = t_i(\overline{X})$$

for $i \in I$ with $\overline{X} = X_1, ..., X_n$ and all $t_i(\overline{X})$ of the form S, D, or $X_{i_l} \trianglelefteq a_i \trianglerighteq X_{i_r}$ for $i_l, i_r \in I$ and $a_i \in A$. In BTA$^\infty$, finite linear recursive specifications represent continuous operators having as unique fixed points *regular* threads, i.e., threads which can only reach finitely many states.

Example 1 *Let $n > 0$. The regular thread $a^n \circ \mathsf{D}$ is the fixed point for X_1 in the specification*

$$\{X_i = a \circ X_{i+1} \mid i = 1, ..., n\} \cup \{X_{n+1} = \mathsf{D}\}.$$

The regular thread $a^n \circ \mathsf{S}$ is the fixed point for X_1 in

$$\{X_i = a \circ X_{i+1} \mid i = 1, ..., n\} \cup \{X_{n+1} = \mathsf{S}\}.$$

Both these threads are finite.

The infinite regular thread a^∞ is the fixed point for X_1 in the specification $\{X = a \circ X\}$ and corresponds to the projective sequence $(P_n)_{n \in \mathbb{N}}$ with $P_0 = \mathsf{D}$ and $P_{n+1} = a \circ P_n$.

Observe that e.g. $a^n \circ \mathsf{D} \sqsubseteq a^n \circ \mathsf{S}$, $a^n \circ \mathsf{D} \sqsubseteq a^\infty$ but $a^n \circ \mathsf{S} \not\sqsubseteq a^\infty$.

For the sake of simplicity, we shall often define regular threads by providing only one or more equations. For example, we say that $P = a \circ P$ defines a regular thread with name P (so $P = a^\infty$ in this case).

We end this section with the observation that for regular threads P and Q, $P \sqsubseteq Q$ is decidable. Because one can always take the disjoint union of two recursive specifications, it suffices to argue that $P_i \sqsubseteq P_j$ in

$$P_1 = t_1(\overline{P}), ..., P_n = t_n(\overline{P})$$

is decidable. This follows from the assertion

$$\forall i, j \leq n \;\; \pi_n(P_i) \sqsubseteq \pi_n(P_j) \Leftrightarrow P_i \sqsubseteq P_j, \tag{1}$$

where $\pi_l(P_k)$ is defined by $\pi_l(t_k(\overline{P}))$, because $\sqsubseteq$ is decidable for finite threads. Without loss of generality, assume $n > 1$. To prove (1), observe that $\Leftarrow$ follows by definition of regular threads. For the reverse, choose i, j and assume that $\pi_n(P_i) \sqsubseteq \pi_n(P_j)$. Suppose $P_i \not\sqsubseteq P_j$, then for some $k > n$, $\pi_k(P_i) \not\sqsubseteq \pi_k(P_j)$ while $\pi_{k-1}(P_i) \sqsubseteq \pi_{k-1}(P_j)$. So there exists a trace of length k from P_i of the form

$$P_i \xrightarrow{a_{\mathsf{true}}} P_{i'} \xrightarrow{b_{\mathsf{false}}} \ldots$$

that is not a trace of P_j, while by the assumption the first n actions are a trace of P_j. These n actions are connected by $n + 1$ states, and because there are only n different states P_1, ...,P_n, a repetition occurs in this sequence of states. So the trace witnessing $\pi_k(P_i) \not\sqsubseteq \pi_k(P_j)$ can be made shorter, contradicting

k's minimality and hence the supposition. Thus $P_i \sqsubseteq P_j$. As a consequence, also $P = Q$ (i.e., $P \sqsubseteq Q$ and $Q \sqsubseteq P$) is decidable for regular threads P and Q.

2.2 Services

A *service* is a pair $\langle \Sigma, F \rangle$ consisting of a set Σ of so-called *co-actions* and a *reply function* F. This reply function is a mapping that gives for each finite sequence of co-actions from Σ a reply value true or false. Services were introduced in [10] under the name "state machines".

Example 2 *A* down counter *or* loop counter *is a service* $\mathtt{DC} = \langle \Sigma, F \rangle$ *with* $\Sigma = \{\mathtt{dec}, \mathtt{set}{:}n \mid n \in I\}$ *consisting of the decrease and set co-actions for some* $I \subseteq \mathbb{N}$ *and the reply function* F *which replies* true *to* $\mathtt{set}{:}n$ *while setting* DC *to value* n*, and* true *to* dec *if* DC*'s value is positive while decreasing its current value, and* false *to* dec *if and only if the counter is zero. The initial value of* DC *is zero and usually* I *will be an initial segment of* $\mathbb{N}$*.*

Down counters (also known as timer units) are crucial components of most embedded systems and included in many microcontrollers (see e.g. [2]). Below, we return to this example.

In order to provide a specific description of the interaction between a thread and a service, we will use for actions the general notation $c.a$ where c is the so-called *channel* or *focus*, and a is the co-action. For example, $c.\mathtt{inc}$ is the action which increases a counter via channel c. This interaction is is defined with help of the *use operator* $/$. For a service $\mathcal{S} = \langle \Sigma, F \rangle$, a finite thread P and a channel c, the defining rules for $P/_c\,\mathcal{S}$ (the thread P *using the service* $\mathcal{S}$ *via channel* c) are:

$$
\begin{aligned}
\mathsf{S}/_c\,\mathcal{S} &= \mathsf{S},\\
\mathsf{D}/_c\,\mathcal{S} &= \mathsf{D},\\
(P \trianglelefteq c'.a \trianglerighteq Q)/_c\,\mathcal{S} &= (P/_c\,\mathcal{S}) \trianglelefteq c'.a \trianglerighteq (Q/_c\,\mathcal{S}) \text{ if } \mathsf{c}' \neq \mathsf{c},\\
(P \trianglelefteq c.a \trianglerighteq Q)/_c\,\mathcal{S} &= P/_c\,\mathcal{S}' \text{ if } a \in \Sigma \text{ and } F(a) = \mathsf{true},\\
(P \trianglelefteq c.a \trianglerighteq Q)/_c\,\mathcal{S} &= Q/_c\,\mathcal{S}' \text{ if } a \in \Sigma \text{ and } F(a) = \mathsf{false},\\
(P \trianglelefteq c.a \trianglerighteq Q)/_c\,\mathcal{S} &= \mathsf{D} \text{ if } a \notin \Sigma.
\end{aligned}
$$

where $\mathcal{S}' = \langle \Sigma, F' \rangle$ with $F'(\sigma) = F(a\sigma)$ for all co-action sequences $\sigma \in \Sigma^+$. The use operator is expanded to infinite threads P by stipulating

$$P/_c\,\mathcal{S} = (\pi_n(P)/_c\,\mathcal{S})_{n \in \mathbb{N}}.$$

As a consequence, $P/_c\,\mathcal{S} = \mathsf{D}$ if for any n, $\pi_n(P)/_c\,\mathcal{S} = \mathsf{D}$. Of course, repeated applications of the use operator bind to the left, thus

$$P/_{c0}\,\mathcal{S}_0/_{c1}\,\mathcal{S}_1 = (P/_{c0}\,\mathcal{S}_0)/_{c1}\,\mathcal{S}_1.$$

We end this section with an example on the use of a service, showing that non-regular threads can be specified with infinite state services.

Example 3 *We may extend the down counter defined in Example 2 to a* full counter $\mathtt{C}$ *by including co-actions* $\mathtt{inc}$ *(increase) which always yield reply* $\mathtt{true}$ *while increasing the counter value. Now let* $\{a, b, c.\mathtt{inc}, c.\mathtt{dec}\} \subseteq A$*. We write* $\mathtt{C}(n)$ *for a counter with value* $n \in \mathbb{N}$*, so* $\mathtt{C} = \mathtt{C}(0)$*. By the defining equations for the use operator it follows that for any thread* P*,*

$$(c.\mathtt{inc} \circ P)/_c\,\mathtt{C}(0) = P/_c\,\mathtt{C}(1),$$

and $\forall n \in \mathbb{N}$*,* $(c.inc \circ P)/_c\,\mathtt{C}(n) = P/_c\,\mathtt{C}(n+1)$*. Furthermore, it easily follows that*

$$(P \trianglelefteq c.\mathtt{dec} \trianglerighteq \mathsf{S})/_c\,\mathtt{C}(n) = \begin{cases} \mathsf{S} & \text{if } n = 0,\\ P/_c\,\mathtt{C}(n-1) & \text{otherwise.}\end{cases}$$

Now consider the regular thread Q *defined by*[1]

$$\begin{aligned} Q &= (c.\mathtt{inc} \circ Q) \trianglelefteq a \trianglerighteq R,\\ R &= b \circ R \trianglelefteq c.\mathtt{dec} \trianglerighteq \mathsf{S}. \end{aligned}$$

Then

$$\begin{aligned} Q/_c\,\mathtt{C}(0) &= ((c.\mathtt{inc} \circ Q) \trianglelefteq a \trianglerighteq R)/_c\,\mathtt{C}(0)\\ &= (Q/_c\,\mathtt{C}(1)) \trianglelefteq a \trianglerighteq (R/_c\,\mathtt{C}(0), \end{aligned}$$

and for all $n \in \mathbb{N}$*,* $Q/_c\,\mathtt{C}(n) = (Q/_c\,\mathtt{C}(n+1)) \trianglelefteq a \trianglerighteq (R/_c\,\mathtt{C}(n)$*. It is not hard to see that* $Q/_c\,\mathtt{C}(0)$ *is an infinite thread with the property that for all* n*, a trace of* $n+1$ a*-actions produced by* n *positive and one negative reply on* a *is followed by* $b^n \circ \mathsf{S}$*. This yields an* irregular *thread: if* $Q/_c\,\mathtt{C}(0)$ *were regular, it would be a fixed point of some finite linear recursive specification, say with k equations. But specifying a trace* $b^k \circ \mathsf{S}$ *already requires* $k+1$ *linear equations* $X_1 = b \circ X_2, ..., X_k = b \circ X_{k+1}, X_{k+1} = \mathsf{S}$*, which contradicts the assumption. So* $Q/_c\,\mathtt{C}(0)$ *is not regular.*

[1] Note that a *linear* recursive specification of Q requires (at least) five equations.

3 Programs and Program Algebra

In this section we introduce the program algebra PGA (see [5]) and discuss its relation with thread algebra. Furthermore, we shortly discuss the unit instruction operator.

3.1 PGA, basics of program algebra

Given a thread algebra with actions in A, we now consider the actions as so-called *basic instructions*. The syntax of PGA has the following primitive instructions as constants:

Basic instruction $a \in A$. It is assumed that upon the execution of a basic instruction, the (executing) environment provides an answer true or false. However, in the case of a basic instruction, this answer is not used for program control. After execution of a basic instruction, the next instruction (if any) will be executed; if there is no next instruction, inaction will occur.

Positive/negative test instruction $\pm a$ for $a \in A$. A positive test instruction $+a$ executes like the basic instruction a. Upon false, the program skips its next instruction and continues with the instruction thereafter; upon true the program executes its next instruction. For a negative test instruction $-a$, this is reversed: upon true, the program skips its next instruction and continues with the instruction thereafter; upon false the program executes its next instruction. If there is no subsequent instruction to be executed, inaction occurs.

Termination instruction !. This instruction prescribes successful termination.

Jump instruction $\#k$ ($k \in \mathbb{N}$). This instruction prescribes execution of the program to jump k instructions forward; if there is no such instruction, inaction occurs. In the special case that $k = 0$, this prescribes a jump to the instruction itself and inaction occurs, in the case that $k = 1$ this jump acts as a *skip* and the next instruction is executed. In the case that the prescribed instruction is not available, inaction occurs.

PGA-terms are composed by means of *concatenation*, notation $\_;\_$, and *repetition*, notation $(\_)^\omega$. Instruction sequence congruence for PGA-terms is axiomatized by the axioms PGA1-4 in Table 1. Here PGA2 is an axiom-*scheme*: for each $n > 0$, $(X^n)^\omega = X^\omega$, where $X^1 = X$ and $X^{k+1} = X; X^k$. A closed PGA-term is often called a PGA-program.

From the axioms PGA1-4 one easily derives *unfolding*, i.e.,

$$X^\omega = X; X^\omega.$$

Table 1.
Axioms for PGA's instruction sequence congruence

$(X;Y);Z = X;(Y;Z)$	(PGA1)	$X^\omega;Y = X^\omega$	(PGA3)
$(X^n)^\omega = X^\omega$ for $n > 0$	(PGA2)	$(X;Y)^\omega = X;(Y;X)^\omega$	(PGA4)

Furthermore, each PGA-program can be rewritten into an instruction equivalent *canonical form*, i.e., a closed term of the form X or $X;Y^\omega$ with X and Y not containing repetition. This also follows from the axioms in Table 1.

We will often use basic instructions in so-called *focus.method* notation, i.e., basic instructions of the form

$$f.m$$

where f is a focus (channel name) and m a method name. The m here is sometimes called a *service-instruction* because it refers to the use of some service, and is related with a co-action as defined in Section 2.2. Two examples of instructions in focus.method notation are $c.\mathtt{inc}$ and $c.\mathtt{dec}$, related with the actions controlling a counter discussed in Example 3. In the next section we will relate all basic and test instructions to the actions of a thread; this is called *thread extraction*.

3.2 Thread extraction: from PGA to thread algebra

The *thread extraction* operator $|X|$ assigns a thread to program object X. Thread extraction is defined by the thirteen equations in Table 2, where $a \in A$ and u is a primitive instruction.

Table 2.
Equations for thread extraction on PGA

$|!| = \mathsf{S}$	$|!;X| = \mathsf{S}$	$|\#k| = \mathsf{D}$
$|a| = a \circ \mathsf{D}$	$|a;X| = a \circ |X|$	$|\#0;X| = \mathsf{D}$
$|{+}a| = a \circ \mathsf{D}$	$|{+}a;X| = |X| \trianglelefteq a \trianglerighteq |\#2;X|$	$|\#1;X| = |X|$
$|{-}a| = a \circ \mathsf{D}$	$|{-}a;X| = |\#2;X| \trianglelefteq a \trianglerighteq |X|$	$|\#k+2;u| = \mathsf{D}$
		$|\#k+2;u;X| = |\#k+1;X|$

Some examples:

$$
\begin{aligned}
|(\#0)^\omega| &= |\#0;(\#0)^\omega| = \mathsf{D},\\
|-a;b;c| &= |\#2;b;c| \trianglelefteq a \trianglerighteq |b;c|\\
&= |\#1;c| \trianglelefteq a \trianglerighteq b \circ |c|\\
&= |c| \trianglelefteq a \trianglerighteq b \circ c \circ \mathsf{D}\\
&= c \circ \mathsf{D} \trianglelefteq a \trianglerighteq b \circ c \circ \mathsf{D}.
\end{aligned}
$$

In some cases, these equations can be applied from left to right without ever generating any behavior, e.g.,

$$|(\#2;a)^\omega| = |\#2;a;(\#2;a)^\omega| = |\#1;(\#2;a)^\omega| = |(\#2;a)^\omega| = \ldots$$

In such cases, the extracted thread is defined as D.

It is also possible that thread extraction yields an infinite recursion, e.g.,

$$|a^\omega| = |a;a^\omega| = a \circ |a^\omega|$$

(in the previous section we denoted this thread by a^∞). If the behavior of X is infinite, it is regular and can be represented by a (linear) recursive specification, e.g.,

$$
\begin{aligned}
|(a;+b;\#3;-b;\#4)^\omega| = P \text{ in } P &= a \circ (P \trianglelefteq b \trianglerighteq Q),\\
Q &= P \trianglelefteq b \trianglerighteq Q.
\end{aligned}
$$

It follows easily that any PGA-program defines a regular thread, and conversely, each regular thread can be defined in PGA: linear equations of the form $X = \mathsf{S}$ or $X = \mathsf{D}$ can be defined by instructions ! and #0, respectively, and a linear equation

$$X = Y \trianglelefteq a \trianglerighteq Z$$

can be associated with a triple $+a;\#k;\#l$. Connecting these program fragments in a repetition and instantiating the jump counters k and l with the appropriate values then yields a PGA-program that defines a solution for the first equation. A typical example:

$$
\begin{array}{lcl}
P_1 = P_2 \trianglelefteq a \trianglerighteq P_2, & & (+a;\#2;\#1;\\
P_2 = P_3 \trianglelefteq b \trianglerighteq P_1, & \mapsto & +b;\#2;\#2;\\
P_3 = \mathsf{D}. & & \#0)^\omega.
\end{array}
$$

For PGA-programs X and Y we write

$$X =_{be} Y$$

if X and Y are behaviorally equivalent (i.e., have the same behavior). Behavior equivalence is not a congruence, e.g., $\#0 =_{be} \#1$ but $\#0;a \neq_{be} \#1;a$. Finally, for a PGA-program X we define

$$X/_c\,\mathcal{S}$$

as the program with behavior $|X|/_c\,\mathcal{S}$, thus $|X/_c\,\mathcal{S}| = |X|/_c\,\mathcal{S}$.

3.3 PGAu, PGA with unit instruction

In [5] the *unit instruction operator*, notation $\mathtt{u}(\_)$ is introduced. This operator wraps a program fragment into a single unit: if X is a program, then $\mathtt{u}(X)$ is a unit that upon execution behaves as X, but that counts as a single instruction in any context. A typical example is

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

which behaves as

$$|b^\omega| \trianglelefteq a \trianglerighteq |c|.$$

A PGA-program that defines the same thread as the above example is for instance

$$+a; (\#2; \#3; b; \#3; c; \#0)^\omega.$$

Typically, a jump to a non-starting position in a unit is not possible, while a jump out of a unit can occur in any position of its body. As an example,

$$+a; \#3; \mathtt{u}(+b; \#3; c); d; e$$

defines the same thread as $+a; \#5; +b; \#3; c; d; e$, i.e.,

$$|e| \trianglelefteq a \trianglerighteq (|e| \trianglelefteq b \trianglerighteq |c; d; e|).$$

Incorporating the unit instruction operator in PGA, notation PGAu, does not increase the expressive power. In this paper we shall make a modest use of the unit instruction operator and we refrain from describing the projection semantics for PGAu as defined in [14]. [2]

The projection semantics for PGAu is defined by a projection function `pgau2pga` (in [14]) on first canonical PGAu-forms, i.e., closed terms of the form X or $X; Y^\omega$ with X and Y not containing repetition. In the particular case that a

[2] This formal semantics is implemented in the PGA Toolset [12] and — including an application of "jump-optimization" — yields for the examples above

$+a; (\#2; \#3; b; \#5; c; \#0)^\omega$, and
$+a; (\#5; +b; \#3; c; d; e; \#0)^\omega$ respectively.

program contains no units, these are first canonical forms in PGA. Furthermore, the projection pgau2pga yields in all cases PGA-programs of the form $(u_1; ...; u_k)^\omega$ and has definitional status. Consequently, each PGAu-program — and therefore each PGA-program — can be expressed in this form. In the next section we will use this property for PGA extended with rigid loops.

4 PGA with rigid loops

In this section we add two types of non-primitive instructions to PGA, thus obtaining PGA with rigid loops. Then we discuss a projection semantics that maps programs to PGAu using counters. We postulate that this semantics has definitional status and argue that this is a reasonable proposal by discussing a "pure projection". Finally, we consider some degenerate examples.

4.1 PGArl, PGA with rigid loops

We add two types of non-primitive instructions to PGA, thus obtaining PGArl, i.e., PGA with rigid loops:

Rigid loop header instruction nx{ for each $n \in \mathbb{N} \setminus \{0\}$. Examples are 7x{ and 432x{. This instruction prescribes an n times repeated execution of the program fragment until the following complementary rigid loop closure instruction. During execution of the body, jumps out of it are permitted and will end its execution; termination within a loop entails termination of the whole program and so does a livelock (#0). A jump into the body of a rigid loop prescribes the execution of its remaining instructions.

Rigid loop closure instruction }x. This instruction ends the body of a rigid loop.

The idea is that the matching of header and closure instructions is innermost-outermost: instruction sequences are parsed left-to-right, so a closure instruction matches the last preceding rigid loop header.

The semantics of PGArl is given by a projection which makes use of an intermediate stage involving annotated closure instructions for rigid loops and annotated jumps out of rigid loops.

Annotated rigid loop closure instruction n}xm for each n and $m \in \mathbb{N}$. This instruction ends the body of a rigid loop with counter value $n+1$ of which the body has a size of m instructions. Its execution is best explained in the presence of a separate loop counter RLC (cf. Example 2) which is initialised at n before execution of the rigid loop and records the number of repetitions still to be done. Executing the annotated closure instruction then consists of #1 if the loop counter RLC has reached value 0 and otherwise a jump to the first instruction of the loop body. These activities must be packed into a single unit in order to preserve the validity of other jumps elsewhere in the program.

In the case that there is no associated rigid loop header instruction, the annotation is $0\}\mathtt{x}0$.

Annotated jump instruction $\#l(j_1,n_1)(j_2,n_2)...(j_k,n_k)$ with $j_i,n_i,k \in \mathbb{N}$ for a jump $\#l$ that jumps over k annotated closure instructions $n_1\}\mathtt{x}m_1,...,n_k\}\mathtt{x}m_k$ at positions $j_1,...,j_k$. The annotation will be used to reset all concerning loop counters.

As an example, $3\mathtt{x}\{;a;b;4\mathtt{x}\{;+c;\#4;\}\mathtt{x};d;\}\mathtt{x};+e;\#3$ yields the annotation

$$3\mathtt{x}\{;a;b;4\mathtt{x}\{;+c;\#4(7,3)(9,2);3\}\mathtt{x}2;d;2\}\mathtt{x}7;+e;\#3.$$

We start with the case that a PGArl-program is of the form

$$(u_1;...;u_k)^\omega,$$

a form which easily facilitates a backward jump to the first instruction of the body of a rigid loop. We adopt the following restrictions on $(u_1;...;u_k)^\omega$:

- each rigid loop header instruction has a complementary closure instruction,
- for each jump instruction $\#m$ it holds that $m < k$ (if not, subtract k sufficiently often),
- rigid loop closures are not preceded by a test instruction.

For the projection we need first to add the annotations, and then to introduce a service for a loop counter attached to each annotated rigid loop closure instruction. The closure instruction at position i will make use of service $\mathtt{rlc}{:}i$. A loop counter has methods $\mathtt{set}{:}n$ which initialises it to n and dec which subtracts 1 if possible while returning a reply true and otherwise returns the reply false.

The projected program begins with an initialisation instruction $\mathtt{rlc}{:}i.\mathtt{set}{:}c_i$ where c_i is the left annotation of the annotated loop closure instruction for each rigid loop that occurs in the candidate program. The loop headers are projected to #1 and their only role has been to determine the annotations for the closure instructions. Thus, assuming that $u_1;...;u_k$ contains l rigid loops with annotated closure instructions at positions $i_1,i_2,...,i_l$, we define

$$\begin{aligned}\mathtt{pgarl2pgau}((u_1;...;u_k)^\omega) = {} & \mathtt{rlc}{:}i_1.\mathtt{set}{:}c_{i1};\mathtt{rlc}{:}i_2.\mathtt{set}{:}c_{i2};...;\mathtt{rlc}{:}i_l.\mathtt{set}{:}c_{il};\\ & (\psi_1(u_1);...;\psi_k(u_k))^\omega/_{\mathtt{rlc}:i_1}\,\mathrm{RLC}_{i_1}/_{\mathtt{rlc}:i_2}\,\mathrm{RLC}_{i_2}.../_{\mathtt{rlc}:i_l}\,\mathrm{RLC}_{i_l}\end{aligned}$$

with

$$
\begin{aligned}
\psi_i(n\texttt{x}\{) &= \#1,\\
\psi_i(\#l(j_1,n_1)...(j_m,n_m)) &= \texttt{u}(\,\texttt{rlc}{:}j_1.\texttt{set}.n_1;\\
&\qquad ...\\
&\qquad \texttt{rlc}{:}j_m.\texttt{set}.n_m;\#l),\\
\psi_i(n\}\texttt{x}m) &= \texttt{u}(\,+\texttt{rlc}{:}i.\texttt{dec};\#3;\\
&\qquad \texttt{rlc}{:}i.\texttt{set}{:}n;\#2;\\
&\qquad \#k-m),\\
\psi_i(u) &= u \text{ otherwise.}
\end{aligned}
$$

Note that in case $(u_1;...;u_k)^\omega$ does not contain rigid loop instructions, we have by definition that $\texttt{pgarl2pgau}((u_1;...;u_k)^\omega) = (u_1;...;u_k)^\omega$.

As a first example, $(3\texttt{x}\{;a;b;4\texttt{x}\{;c;\}\texttt{x};d;\}\texttt{x};e)^\omega$ yields the annotated program

$$(3\texttt{x}\{;a;b;4\texttt{x}\{;c;3\}\texttt{x}1;d;2\}\texttt{x}6;e)^\omega,$$

which yields under `pgarl2pgau`

$$
\begin{aligned}
&\texttt{rlc}{:}6.\texttt{set}{:}2;\texttt{rlc}{:}8.\texttt{set}{:}3;\\
&(\#1;a;b;\#1;c;\ \texttt{u}(\,+\texttt{rlc}{:}6.\texttt{dec};\#3;\\
&\qquad\qquad\qquad \texttt{rlc}{:}6.\texttt{set}{:}2;\#2;\\
&\qquad\qquad\qquad \#8);\\
&\ d;\ \texttt{u}(\,+\texttt{rlc}{:}8.\texttt{dec};\#3;\\
&\qquad \texttt{rlc}{:}8.\texttt{set}{:}3;\#2;\\
&\qquad \#3);\\
&\ e)^\omega/_{\texttt{rlc}:6}\,\texttt{RLC}_6/_{\texttt{rlc}:8}\,\texttt{RLC}_8
\end{aligned}
$$

and thus defines the thread P given by $P = (a \circ b \circ c^4 \circ d)^3 \circ e \circ P$.

As a second example, consider the program $(a;2\texttt{x}\{;+b;\#3;\}\texttt{x};c;d)^\omega$, thus

$$(a;2\texttt{x}\{;+b;\#3(5,1);1\}\texttt{x}2;c;d)^\omega,$$

which has the option of ending a rigid loop by jumping out of it: under `pgarl2pgau` we obtain

$\texttt{rlc:5.set:1};$
$(a;\#1;+b;\ \texttt{u}(\texttt{rlc:5.set:1};\#3);$

$\texttt{u}(\ +\texttt{rlc:5.dec};\#3;$
$\texttt{rlc:5.set:1};\#2;$
$\#5);$
$c;d)^{\omega}/_{\texttt{rlc:5}}\,\texttt{RLC}_5$

which defines the thread P given by

$$P = a \circ (d \circ P \trianglelefteq b \trianglerighteq (d \circ P \trianglelefteq b \trianglerighteq c \circ d \circ P)).$$

For a repetition-free PGArl-program $u_1;...;u_k$ we define

$$\texttt{pgarl2pgau}(u_1;...;u_k) = \texttt{pgarl2pgau}(\Phi(u_1;...;u_k)),$$

where the transformation Φ is given by

$$\Phi(u_1;...;u_k) = (\phi_1(u_1);...;\phi_k(u_k);\#0;\#0)^{\omega},$$
$$\phi_i(\#n) = \#\min(n, k+2-i),$$
$$\phi_i(u) = u \text{ otherwise.}$$

Here the latter two #0-instructions serve the case that u_k is a test instruction.

It remains to define the projection pgarl2pgau for first canonical forms

$$u_1;...;u_k;(v_1;...;v_l)^{\omega}$$

with $k, l > 0$. In this case we may assume that if $u_i = \#m$, then $m \leq k - i + l$ (otherwise, subtract l sufficiently often). Similarly, we may assume that if $v_j = \#m$, then $m < l$. We define

$$\texttt{pgarl2pgau}(u_1;...;u_k;(v_1;...;v_m)^{\omega}) = \texttt{pgarl2pgau}(\Xi(u_1;...;u_k;(v_1;...;v_m)^{\omega}))$$

with

$$\Xi(u_1;...;u_k;(v_1;...;v_m)^{\omega}) = (u_1;...;u_k;\xi_1(v_1);...;\xi_m(v_m);\#k;\#k)^{\omega},$$
$$\xi_i(\#n) = \#n + k + 2 \text{ if } i + n > m,$$
$$\xi_i(u) = u \text{ otherwise.}$$

This completes the definition of pgarl2pgau and we give this projection *definitional status*. In other words, the loop counter service based projection pgarl2pgau is the *defining projection* for PGArl.

4.2 Pure projection of rigid loops and definitional status

In the previous section we assumed that PGArl-programs satisfy a certain well-formedness criterion:

- each rigid loop header instruction has a complementary closure instruction,
- for each jump instruction $\#m$ in $(u_1; ...; u_k)^\omega$ it holds that $m < k$ (if not, subtract k sufficiently often),
- rigid loop closures are not preceded by a test instruction.

Before dealing with programs that are not well-formed, we first discuss pure projection of well-formed PGArl-programs.

The pure PGA projection `pgarl2pga` expands the body of each loop while adapting appropriately the jumps that go into the body and that might exit from the body. Expansion can be defined in a left-to-right order on rigid loop headers in the following way: let X be a (possibly empty) sequence of PGA-instructions, u_i range over the PGArl-instructions, and let Y range over finite (possibly empty) sequences of PGArl-instructions. Then

$$X; 1\mathtt{x}\{; u_1; ...; u_k; \}\mathtt{x}; Y = X; \#1; u_1; ...; u_k; \#1; Y \tag{2}$$

and for all $n > 1$,

$$X; (n+1)\mathtt{x}\{; u_1; ...; u_k; \}\mathtt{x}; Y = X'; \#1; u'_1; ...; u'_k; \#1; n\mathtt{x}\{; u_1; ...; u_k; \}\mathtt{x}; Y \tag{3}$$

where

$$u'_i = \begin{cases} \#m + k + 2 & \text{if } u_i = \#m \text{ and } i + m > k + 1, \\ u_i & \text{otherwise,} \end{cases}$$

$X' = X$, except that all jumps in X that pass $(n+1)\mathtt{x}\{; u_1; ...; u_k; \}\mathtt{x}$ are raised with $k + 2$.

With these two equations all rigid loops can be removed in $(u_1; ...; u_k)^\omega$, and defining

$$\mathtt{pgarl2pga}(X) = X \quad \text{if } X \text{ is a PGA-program}$$

completes the definition of this pure projection.

We first argue that the expansion equation (3) is sound for the finite case. Let

$$t_1 = v_1; ...; v_r; (n+1)\mathtt{x}\{; u_1; ...; u_k; \}\mathtt{x}; w_1; ...; w_s,$$
$$t_2 = v'_1; ...; v'_r; \#1; u'_1; ...; u'_k; \#1; n\mathtt{x}\{; u_1; ...; u_k; \}\mathtt{x}; w_1; ...; w_s.$$

We show that $\texttt{pgarl2pgau}(t_1) =_{be} \texttt{pgarl2pgau}(t_2)$ by case distinction on the various instructions in t_1, assuming t_1 contains l rigid loops with their closure instructions at positions $i_1, ..., i_l$ (so $i_1 = r + k + 2$). Without loss of generalization we further assume that jumps outside the program are such that in $t_1; \#0; \#0$ they end in one of the latter two $\#0$ instructions, and thus we can and will leave out the repetition in $\texttt{pgarl2pgau}(t_1)$. By a similar argument, the repetition in $\texttt{pgarl2pgau}(t_2)$ is left out.

With respect to the instructions v_i, the only interesting case is $v_i = \#j$ with $i + j > r$. We distinguish four sub-cases:

(a) If $i+j = r+1$, this prescribes a jump (via $\#1$) to the instruction $\psi_{r+2}(u_1)$. In t_2's projection there is a jump to the instruction $\psi_{r+2}(u'_1)$. We proceed with this case below.

(b) If $r + 1 < i + j < r + k + 2$, then $\psi_q(u_p)$ in t_1's projection and the associated $\psi_q(u'_p)$ in t_2's projection have to be related. We proceed with this case below.

(c) If $i + j = r + k + 2$, then $\texttt{pgarl2pgau}(t_1)$ is further determined by

$$\begin{array}{l}\#1; \psi_{r+2}(u_1); ...; \psi_{r+k+1}(u_k); \texttt{u}(...);\\ \psi_{r+k+3}(w_1); ...; \psi_{r+k+s+2}(w_s); \#0; \#0/_{\texttt{rlc}:r+k+2}\, \texttt{RLC}_{r+k+2}(n-1)...\end{array}$$

and so is $\texttt{pgarl2pgau}(t_2)$ (although all its ψ-indices and foci- and counter-indices are raised with $k + 2$, but this is not significant). So in this case, $\texttt{pgarl2pgau}(t_1) =_{be} \texttt{pgarl2pgau}(t_2)$.

(d) If $i + j > r + k + 2$, then in both $\texttt{pgarl2pgau}(t_1)$ and $\texttt{pgarl2pgau}(t_2)$ this prescribes a jump to the w-part or to one of the two added $\#0$'s and $\texttt{RLC}_{r+k+2}$ respectively $\texttt{RLC}_{r+2k+4}$ do not play a role, so also in this case behavioral equivalence holds.

According to the first two cases it remains to be proved that

$$\begin{array}{l}\psi_{r+i+1}(u_i); ...; \psi_{r+k+1}(u_k); \texttt{u}(...);\\ \psi_{r+k+3}(w_1); ...; \psi_{r+k+s+2}(w_s); \#0; \#0/_{\texttt{rlc}:r+k+2}\, \texttt{RLC}_{r+k+2}(n)...\\ =_{be}\\ \psi_{r+i+1}(u'_i); ...; \psi_{r+k+1}(u'_k); \#1; \#1; \psi_{r+k+4}(u_1); ...; \psi_{r+2k+3}(u_k); \texttt{u}(...);\\ \psi_{r+2k+5}(w_1); ...; \psi_{r+2k+s+4}(w_s); \#0; \#0/_{\texttt{rlc}:r+2k+4}\, \texttt{RLC}_{r+2k+4}(n-1)...\end{array} \quad (4)$$

for $i = 1, ..., k$. We discuss the following cases:

(e) If $u_i = \#j$ and $i + j = k + 1$, then in the lhs above the rigid loop is restarted at its first instruction with counter value $n - 1$, and so happens in the rhs, so the behavioral equivalence in (4) holds.

(f) If $u_i = \#j$ and $i + j > k + 1$, this prescribes in both sides a jump to the w-part or to one of the added $\#0$'s, and the behavioral equivalence in (4) holds.

(g) If $u_i = m\mathtt{x}\{$ (and its closure instruction is in the u-part), then in both the lhs and the rhs that rigid loop is either completed and behavior proceeds while the index i in (4) has raised, or the loop is jumped out and the resulting position either matches one of the two cases above, or is into the u-part. In the latter case, also the index i in (4) has raised.

It follows that for all instantiations of u_i we either obtain the behavioral equivalence in (4), or the index i raises until we are at least at position $r+k+2$ and behavioral equivalence then follows from the sub-cases (c) and (d) above. This completes our argument on the soundness of equation (3) for the finite case. A comparable, but more simple analysis reveals the soundness of equation (2) for finite PGArl-programs.

The iterative case is slightly more complex, as jumps can have a backward target. However, a similar analysis shows that also in this case both equations (2) en (3) are sound. This completes our argument on the soundness of the pure projection `pgarl2pga`.

The pure projection clearly provides a combinatorial explosion. It can be concluded that the loop counter service based projection `pgarl2pgau` is indeed the best candidate for a defining semantics: it satisfies both the criterion *normative semantic adequacy* and the criterion *indicative algorithmic adequacy* while the pure projection satisfies only the first one.

The projection `pgarl2pgau` *defines* the meaning of rigid loop instructions also for the degenerate case that a rigid loop header instruction has no associated closure instruction or vice versa: such a lonely instruction acts as a skip (i.e., #1). Finally, note that a rigid loop body of length 0 is unproblematic: it has no behavioral impact (of course, this holds as well for the pure projection).

5 Conclusions

First we note that the defining projection `pgarl2pgau` uses finite state services. Indeed, any PGArl-program not containing repetition can be expanded to one without rigid loops (using the expansion equations (2) and (3)).

Although rigid loops are less expressive than arbitrary loops and fail to express all finite state threads they can be proven sufficient for programming state transformations on finite Maurer computers (see [7–9]). Admittedly one may be forced into using quite large loop counters but in principle it works.

Acknowledgements

We thank Bob Diertens for valuable remarks.

References

[1] J.W. de Bakker and J.I. Zucker. Processes and the denotational semantics of concurrency. *Information and Control*, 54(1/2):70–120, 1982.

[2] S. Ball and M. Barr. Introduction to Counter/Timer Hardware. *Embedded Systems Programming*, September 2002, pp. 55-56, 2002.

[3] J.A. Bergstra and I. Bethke. Polarized process algebra and program equivalence. In J.C.M. Baeten, J.K. Lenstra, J. Parrow, and G.J. Woeginger, editors, *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4*, Springer-Verlag, LNCS 2719:1-21, 2003.

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

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

[6] J.A. Bergstra and C.A. Middelburg. A thread algebra with multi-level strategic interleaving. In S.B. Cooper, B. Loewe and L. Torenvliet, editors, CiE 2005, Springer-Verlag, LNCS 3526:35-48, 2005.

[7] J.A. Bergstra and C.A. Middelburg. Maurer computers with single-thread control. Computer Science Report 05-17, Department of Mathematics and Computer Science, Eindhoven University of Technology, June 2005.

[8] J.A. Bergstra and C.A. Middelburg. Simulating Turing machines on Maurer machines. Computer Science Report 05-28, Department of Mathematics and Computer Science, Eindhoven University of Technology, November 2005.

[9] J.A. Bergstra and C.A. Middelburg. Maurer computers for pipelined instruction processing. Computer Science Report 06-12, Department of Mathematics and Computer Science, Eindhoven University of Technology, March 2006.

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

[11] A. Bolychevsky, C. R. Jesshope, and V. Muchnick. Dynamic scheduling in RISC architectures. *IEE Proceedings Computers and Digital Techniques*, 143(5):309-317, 1996.

[12] B. Diertens. *PGA - ProGram Algebra*. Web site offering a Toolset for PGA: `www.science.uva.nl/research/prog/projects/pga/`, Last modified: July 03, 2006.

[13] C.R. Jesshope and B. Luo. Micro-threading: A new approach to future RISC. In *ACAC 2000*, pages 34-41. IEEE Computer Society Press, 2000.

[14] A. Ponse. Program algebra with unit instruction operators. *Journal of Logic and Algebraic Programming*, 51(2):157–174, 2002.