Bachelor Informatica

University of Amsterdam

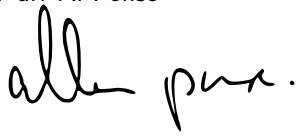
On finite projections and program length in thread and program algebra

M.G. Redder

June 9, 2017

Supervisor: dr. A. Ponse

Signed:



Abstract

In this thesis it is shown that, for any process generated by a recursion variable in a linear recursive specification as specified by Thread Algebra, its behaviour is fully defined within its recursive specification at its $(n-1)^{th}$ projection, where n equals the total number of equations in the recursive specification. This also means that, given the two sets of linear equations expressing the behaviour of any two programs, the equality or inequality of the respective described regular behaviours can be evaluated at the $(n-1)^{th}$ projection, where n equals the total number of linear equations in both sets combined.
Additionally it is shown that, given a program of length m in Program Algebra, the maximum number of equations of the smallest linear specification describing the associated behaviour is $m+1$. Similar maximum numbers can be derived for Program Algebra's higher languages.
In particular, the combination of these two results signifies that for any two programs in PGA consisting of n and m instructions respectively, behavioural equality can be evaluated with certainty by calculating the $(n+m+1)^{th}$ projection of the processes generated by each of the programs.

Contents

CHAPTER 1

Introduction

In order to explore the mathematical properties of computer code, formalisms such as Program Algebra and Thread Algebra are developed to model instruction sequences and the behaviour resulting from the execution of such instruction sequences.
What is commonly referred to as a *program* (or the control structure thereof), can be finitely represented in Program Algebra and gives rise to an instruction sequence, which can be infinite. In this thesis, two mathematical properties of such instruction sequences are explored.

We first look at the relation between the complexity of behaviour associated with the execution of instruction sequences, and finite projections of this behaviour. This is of particular interest when this behaviour is infinite.
We then proceed by exploring how this complexity of behaviour relates to the number of instructions constituting the associated program.

The thesis is constructed in the following way:
In Chapter 2, some preliminaries are discussed. These concepts are used in the subsequent chapters.
In Chapter 3, the first research question is presented and discussed:
"At what projection is the behaviour of a program fully defined within its recursive specification?".
Chapter 4 connects the results from Chapter 3 to the number of instructions in a given program, answering the second research question:
"What is the relation between a program's length and the maximum number of equations needed to minimally express its behaviour?".
Ultimately, in Chapter 5 we state the conclusions that can be drawn from the results of the previous chapters.

CHAPTER 2

Preliminaries

Program Algebra

Syntax

Program Algebra (PGA), as introduced by Bergstra and Loots [3], is an algebra created to model imperative, sequential programming. Its syntax is generated from five kinds of primitive instructions, which can be composed with two methods of composition.

The five kinds of primitive instructions are:

- basic instruction: $a \in \Sigma$
  After execution of basic instruction a, the next instruction should be carried out. If there is no next instruction, inaction occurs.
- termination: !
  Successfully terminates an instruction sequence.
- positive test instruction: $+a$ with $a \in \Sigma$
  Executes action a, after which the next instruction should be carried out if a returned *true* or the one after the next one if a returned *false*. If the appropriate subsequent instruction does not exist, inaction follows.
- negative test instruction: $-a$ with $a \in \Sigma$
  Executes action a, after which the next instruction should be carried out if a returned *false* or the one after the next one if a returned *true*. If the appropriate subsequent instruction does not exist, inaction follows.
- forward jump instruction: $\#k$
  Moves execution to the k^{th} instruction following this jump instruction. In other words: #1 indicates the very next instruction should be executed, whereas #2 indicates the next instruction should be skipped and the one after that one should be carried out. Again, if the required instruction does not exist, inaction follows. A special case is #0: as this indicates the execution of the jump instruction itself, this instruction will be executed again without carrying out any action. Therefore, execution of #0 equals inaction.

In this thesis, capitals X, Y and Z are used as variables for sequences of such instructions, or *programs*. Variables u_i with $i \in \mathbb{N}^+$ are used to denote unspecified primitive instructions.

These primitive instructions can be composed with the two methods of composition:

- concatenation: $X; Y$
  Indicates that instruction sequence X is carried out first, after which instruction sequence Y is executed.

- repetition: X^ω
  Indicates that instruction sequence X is repeated infinitely. After the last instruction of X, its first instruction follows again.

As an example, the instruction sequence $+a; \#0; -b; !; (a; c; \#2)^\omega$ starts by executing positive test instruction $+a$. If a returns *true*, then $\#0$ follows: inaction. However, if a returns *false*, negative test instruction $-b$ is carried out. This executes action b, and if this returns *false*, the program successfully terminates with !. However, if b returns *true*, a is executed, after which c is carried out. Subsequently, jump instruction $\#2$ redirects execution to the 2^{nd} instruction after this jump instruction. Because of the repetition operator $^\omega$ applied to the bracketed sequence $(a; c; \#2)$, the beginning of this bracketed sequence follows again after its last instruction - so the 2^{nd} instruction after $\#2$ is c again. c, in turn, is followed by $\#2$, which directs the execution back to c. So once this part of the program has been reached, c will be carried out infinitely many times.

First and second canonical form

Programs are said to be *instruction sequence equivalent*, if they can be shown to be equal by means of the equations in Table 2.1. In particular, these equations can be used to rewrite any

Table 2.1: Program object equations: PGA1 - PGA4

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

program into of two forms:

- Y, not containing the repetition operator
- $Y; Z^\omega$, with Y and Z themselves not containing the repetition operator

A program written in either one of these two forms, is said to be in *first canonical form*.
Additionally, such a sequence of instructions can be simplified further using the equations in Table 2.2 such that there are no chains of consecutive jump instructions (PGA5 and PGA6), and such that - in case of a canonical form $Y; Z^\omega$ - counters of jump instructions within and into the repeating part Z are as small as possible (PGA7 and PGA8). A program in first canonical form which also adheres to these two additional requirements, is said to be in *second canonical form*.
If two programs can be proven equal by means of PGA1 - PGA8, they are *structurally congruent*.

Table 2.2: PGA5 - PGA8

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

Higher languages

Higher languages with auxilary actions, for example those containing instructions that manipulate nested expressions and therefore require stack manipulation, can be mapped to PGA with so-called projections. Conversely, sequences of PGA instructions can be translated to such higher languages using so-called embeddings, while preserving the original semantics.
We hereby look at two such higher languages: PGLA and PGLB.

PGLA

PGLA is an adaptation of PGA with the additional *repeat instruction* $\backslash\backslash\#k$ for any number $k \in \mathbb{N}^+$. This instruction indicates that the last k instructions should be repeated, as well as the repeat instruction itself. As such, the repeat instruction replaces the repetition operator $^\omega$. Any program in PGA X can be transformed into PGLA by means of its embedding:

- if X is rewritten into a second canonical form and as such is of the form Y without repetition, then PGA2PGLA(Y) = Y.
- if X is rewritten into a second canonical form and as such is of the form

$$Y; Z^\omega \;=\; u_1; ...; u_k; (u_{k+1}; ...; u_{k+m})^\omega,$$

then

$$\text{PGA2PGLA}(u_1; ...; u_k; (u_{k+1}; ...; u_{k+m})^\omega) \;=\; u_1; ...; u_k; u_{k+1}; ...; u_{k+m}; \backslash\backslash\#m$$

For instance:

$$\text{PGA2PGLA}(a; b; (c; d)^\omega) \;=\; a; b; c; d; \backslash\backslash\#2$$

.
Any program in PGLA can be projected directly to a canonical form of PGA:

- If sequence X in PGLA contains no repeat instruction, the projection is given by PGLA2PGA(X) $=$ X.
- If sequence X contains at least one repeat instruction, all instructions to the right of the left-most repeat instruction are removed. Subsequently, the projection to PGA of the remaining expression $X \;=\; u_1; ...; u_k; \backslash\backslash\#m$ depends on the relation between k and m:
  - if $k > m$, then $\text{PGLA2PGA}(X) = u_1; ...; u_{k-m}; (u_{k-m+1}; ...; u_k)^\omega$
  - if $k = m$, then $\text{PGLA2PGA}(X) = u_1; ...; u_k; (u_1; ...; u_k)^\omega$
  - if $k < m$, then $\text{PGLA2PGA}(X) = u_1; ...; u_k; (\#0)^{m-k}; (u_1; ...; u_k; (\#0)^{m-k})^\omega$

For example:

$$\text{PGLA2PGA}(a; b; c; \backslash\backslash\#2) \;=\; a; (b; c)^\omega$$

$$\text{PGLA2PGA}(a; b; c; \backslash\backslash\#3) \;=\; a; b; c; (a; b; c)^\omega$$

$$\text{PGLA2PGA}(a; b; c; \backslash\backslash\#4) \;=\; a; b; c; \#0; (a; b; c; \#0)^\omega$$

PGLB

PGLB is a variation on PGLA containing the backward jump $\backslash\#k$, making the repeat instruction $\backslash\backslash\#k$ redundant.
A program in PGA can be embedded into PGLB by first obtaining a second canonical form using PGA1 - PGA8. The embedding depends on its appearence in second canonical form:

- if X is of the form $X \;=\; Y$ without repetition operator, then $\text{PGA2PGLB}(X) \;=\; X$.
- if X is of the form $X \;=\; Y; Z^\omega \;=\; u_1; ...; u_k; (u_{k+1}; ...; u_{k+m})^\omega$, then $\text{PGA2PGLB}(u_1; ...; u_k; (u_{k+1}; ...; u_{k+m})^\omega) \;=\; u_1; ...; u_k; \vartheta_1(u_{k+1}); ...; \vartheta_m(u_{k+m}); \backslash\#m; \backslash\#m$ with operators ϑ_j defined as:

$$\begin{aligned} \vartheta_j(\#l) \;&=\; \backslash\#m - l && \text{if } j + l > m \\ \vartheta_j(u) \;&=\; u && \text{if otherwise.} \end{aligned}$$

For example:

$$\text{PGA2PGLB}(-a; \#4; (b; c; \#2; +a)^\omega) \;=\; -a; \#4; b; c; \#2; +a; \backslash\#4; \backslash\#4$$

Any instruction sequence in PGLB can be projected to PGA directly by the projection:

$$\text{PGLB2PGA}(u_1;...;u_k) = (\psi_1(u_1);...;\psi_n(u_n);\#0;\#0)^\omega.$$

in which function $\psi_j(u)$ is defined as:

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

For instance:

$$\text{PGLB2PGA}(+a;\#2;+b;\backslash\#5;d;+e;\backslash\#2;\backslash\#5) = (+a;\#2;+b;\#0;d;+e;\#8;\#5;\#0;\#0)^\omega$$

Thread Algebra

The behaviour of programs described in PGA or its higher languages can be expressed in Thread Algebra, such that behavioural equivalence of different sequences of instructions can be determined.

Primitives

Similar to PGA, TA involves the collection Σ of abstract basic instructions, which are referred to as *actions* when it involves the behaviour associated with that particular instruction.
In addition to these basic actions, two other constants are considered primitives:

- S
  Expresses termination behaviour.
- D
  Represents divergence or inaction.

All these primitives can be composed by means of the *postconditional composition operator*:

$$P \trianglelefteq a \trianglerighteq Q$$

This operator indicates that action a is executed, and if this action returns boolean *true*, execution continues with expression P. If a returns boolean *false*, execution continues with expression Q. A special case of postconditional composition occurs for basic instructions that do not appear as a test instruction: instead of writing

$$P \trianglelefteq a \trianglerighteq P$$

to indicate that execution continues with consecutive expression P regardless of the return value of a, the action prefix notation can be used:

$$a \circ P$$

Projective sequences

As expressions in Thread Algebra model finite behaviour, *projective sequences* are used to describe infinite behaviour. These sequences make use of *projection operators* $\pi_n(P)$ with $n \in \mathbb{N}$, each indicating expression of the behaviour of P up until n actions. The projection operators are defined inductively:

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

A projective sequence $(P_n)_{n\in\mathbb{N}}$ is defined as a sequence $P_0, P_1, P_2, ...$ where for each $n \in \mathbb{N}$ two conditions are met:

- P_n is an expression of finite behaviour
- $\pi_n(P_{n+1}) = P_n$

To identify infinite behaviour, equality of infinite behaviours can be determined by means of the corresponding projective sequences:

$$(P_n)_{n\in\mathbb{N}} = (Q_n)_{n\in\mathbb{N}} \quad \text{if} \quad P_n = Q_n \text{ for all } n$$

Extracting program behaviour

To denote extraction of behaviour on a program object X, the notation $|X|$ is used. This extraction is defined by the equations in Table 2.3. Additionally, convention is that the extracted

Table 2.3: Equations for extraction of behaviour from PGA-programs of n instructions

$\|!\| = S$	(2.1)
$\|a\| = a \circ D$	(2.2)
$\|+a\| = a \circ D$	(2.3)
$\|-a\| = a \circ D$	(2.4)
$\|\#k\| = D$	(2.5)
$\|!;X\| = S$	(2.6)
$\|a;X\| = a \circ D$	(2.7)
$\|+a;X\| = \|X\| \trianglelefteq a \trianglerighteq \|\#2;X\|$	(2.8)
$\|-a;X\| = \|\#2;X\| \trianglelefteq a \trianglerighteq \|X\|$	(2.9)
$\|\#0;X\| = D$	(2.10)
$\|\#1;X\| = \|X\|$	(2.11)
$\|\#k+2;u\| = D$	(2.12)
$\|\#k+2;u;X\| = \|\#k+1;X\|$	(2.13)

behaviour in cases of infinite application of equations *not* leading to any behaviour equals D, for example:

$$|(\#n;u_1;...;u_{n-1})^\omega|S = |(\#n)^\omega| = D \tag{2.14}$$

Using these equations and PGA1-PGA4, it is possible to express the behaviour associated with an instruction sequence X as a finite set of recursive equations, each of one of three forms:

$$X_i = X_{i1} \trianglelefteq a_i \trianglerighteq X_{i2} \tag{2.15}$$
$$X_j = S \tag{2.16}$$
$$X_k = D \tag{2.17}$$

For a linear recursive specification E, the notation $\langle X_i|E\rangle$ denotes the process that forms the solution for the equation X_i. If E is known, we simply write $\langle X_i\rangle$ instead of $\langle X_i|E\rangle$.

CHAPTER 3

Finite projections and equality of program behaviour

Research question 1:

"At what finite projection is the behaviour of a program fully defined within its recursive specification?"

In the previous chapter, equality of infinite behaviours was said to be determined by means of the corresponding projective sequences:

$$(P_n)_{n\in\mathbb{N}} = (Q_n)_{n\in\mathbb{N}} \quad \text{if} \quad P_n = Q_n \text{ for all } n$$

In other words: if all finite projections of two processes are equal, the two processes are equal. However, analogue to the results of Barros and Hou [1], in fact not all finite projections need to be compared.

A theorem on finite projections

For any two processes $\langle X_p\rangle$ and $\langle X_q\rangle$, a shared recursive specification E can be obtained by taking the disjoint union of both sets of equations. In this chapter, we show that the behavioural equality of two recursion variables X_p and X_q in $V_E = \{X_1, X_2, ..., X_n\}$ as defined by some linear recursive specification E, can be determined at the $(n-1)^{th}$ projection:

$$\pi_{n-1}(\langle X_p\rangle) = \pi_{n-1}(\langle X_q\rangle) \quad \Rightarrow \quad \langle X_p\rangle = \langle X_q\rangle. \tag{3.1}$$

In order to prove this, we consider the relation $\cong_k$ on $\langle V_E\rangle = \{\langle X\rangle \mid X \in V_E\}$ as defined by Barros and Hou in [1] by:

$$\langle X_p\rangle \cong_k \langle X_q\rangle \Leftrightarrow \pi_k(\langle X_p\rangle) = \pi_k(\langle X_q\rangle).$$

For each number k, $\cong_k$ is an equivalence relation, defining a partition of the set V_E into a certain number of equivalence classes.
Analogous to [1], we now need to prove two claims:
1. Once the partition of V_E defined by $\cong_k$ is equal to the one defined by $\cong_{k+1}$, then this partition remains the same for $\cong_h$ with all subsequent numbers $h > k$.
2. This is the case from $(n-1)$ at most.

Claim 1: $(\cong_k = \cong_{k+1}) \Rightarrow (\cong_{k+1} = \cong_{k+2})$

Since

$$\pi_{k+1}(\langle X_p\rangle) = \pi_{k+1}(\langle X_q\rangle) \Rightarrow \pi_k(\langle X_p\rangle) = \pi_k(\langle X_q\rangle)$$

it must be the case that

$$\cong_{k+1} \quad \supseteq \quad \cong_{k+2}$$

so we only need to prove that

$$\cong_{k+1} \quad \subseteq \quad \cong_{k+2}.$$

Assume the contrary, so

$$\cong_{k+1} \quad \not\subseteq \quad \cong_{k+2}$$

Then there are X_p and X_q such that

$$\pi_{k+1}(\langle X_p \rangle) \;=\; \pi_{k+1}(\langle X_q \rangle) \tag{3.2}$$

but

$$\pi_{k+2}(\langle X_p \rangle) \;\neq\; \pi_{k+2}(\langle X_q \rangle). \tag{3.3}$$

Because of (3.3), we know that X_p and X_q can not *both* be S, nor can they *both* be D. This means at most one of them is S and at most one of them is D.
However, because of (3.2) and because, by definition, $\pi_{k+1}(S) = S$ and $\pi_{k+1}(D) = D$, it can also not be true that one is S and one is D, nor can it be the case that either of them is S or D and the other one is *neither* S nor D. After all, the latter would mean that any projection π_{k+1} of its behaviour would be equal to something of the form $\pi_k(\langle X_i \rangle) \trianglelefteq a \trianglerighteq \pi_k(\langle X_j \rangle)$ for some $a \in A$. As such, it would start with at least one action $a \in A$ and thus be unequal to $\pi_{k+1}(S) = S$ as well as $\pi_{k+1}(D) = D$.
Therefore, X_p and X_q must each consist of an action $a \in A$ in postconditional composition with two (not necessarily unique) variables from V_E.
Now, let

$$X_p \;=\; X_i \trianglelefteq a_p \trianglerighteq X_j$$
$$X_q \;=\; X_l \trianglelefteq a_q \trianglerighteq X_m$$

so

$$\pi_{k+2}(\langle X_p \rangle) \;=\; \pi_{k+1}(\langle X_i \rangle) \trianglelefteq a_p \trianglerighteq \pi_{k+1}(\langle X_j \rangle)$$
$$\pi_{k+2}(\langle X_q \rangle) \;=\; \pi_{k+1}(\langle X_l \rangle) \trianglelefteq a_q \trianglerighteq \pi_{k+1}(\langle X_m \rangle).$$

These are not equal because of (3.3), so

$$a_p \;\neq\; a_q \quad \vee \quad \pi_{k+1}(\langle X_i \rangle) \;\neq\; \pi_{k+1}(\langle X_l \rangle) \quad \vee \quad \pi_{k+1}(\langle X_j \rangle) \;\neq\; \pi_{k+1}(\langle X_m \rangle).$$

Because of the condition ($\cong_k \;=\; \cong_{k+1}$), this implies that

$$a_p \;\neq\; a_q \quad \vee \quad \pi_k(\langle X_i \rangle) \;\neq\; \pi_k(\langle X_l \rangle) \quad \vee \quad \pi_k(\langle X_j \rangle) \;\neq\; \pi_k(\langle X_m \rangle).$$

This means that

$$\pi_{k+1}(\langle X_p \rangle) \;=\; \pi_k(\langle X_i \rangle) \trianglelefteq a_p \trianglerighteq \pi_k(\langle X_j \rangle) \qquad \text{can not be equal to}$$

$$\pi_{k+1}(\langle X_q \rangle) \;=\; \pi_k(\langle X_m \rangle) \trianglelefteq a_q \trianglerighteq \pi_k(\langle X_l \rangle).$$

This is in contradiction with (3.2), and thus contradicts our assumptions. □

Claim 2: $\cong_{n-1} \quad = \quad \cong_n$

If an equivalence relation R is a proper subset of equivalence relation R', then the number of equivalence classes generated by R must be greater than the number of equivalence classes

generated by R'.
The maximum number of equivalence classes of V_E is n (as there are n recursion variables) and the sequence of relations defined by $\cong_k$ is initially strictly decreasing, so at most n relations in this sequence can be unequal: $\cong_0$ to $\cong_{n-1}$.
Therefore it must be the case that $\cong_{n-1} = \cong_n$. □

Conclusion from claim 1 and claim 2

Using the validity of these two claims, we can now conclude that

$$\pi_{n-1}(\langle X_p \rangle) = \pi_{n-1}(\langle X_q \rangle) \quad \text{implies that} \quad \forall k \geq 0 \quad \pi_k(\langle X_p \rangle) = \pi_k(\langle X_q \rangle)$$

and therefore

$$\pi_{n-1}(\langle X_p \rangle) = \pi_{n-1}(\langle X_q \rangle) \quad \Rightarrow \quad \langle X_p \rangle = \langle X_q \rangle$$ □

Tightness

It can be shown that the above is *tight*: in general, the same is not true for any value lower than $n-1$. That is, for any value n equal to or larger than 2, there exists a linear recursive specification E with n equations, and with $X_p, X_q \in V_E$ such that

$$\pi_{n-2}(\langle X_p \rangle) = \pi_{n-2}(\langle X_q \rangle) \text{ and } \langle X_p \rangle \neq \langle X_q \rangle$$

This is proved in [1] for BPA_δ, and the proof immediately transfers to the setting of Thread Algebra.

CHAPTER 4

On program length and minimal specification of program behaviour

Research question 2: *"What is the relation between a program's length and the maximum number of equations needed to minimally express its behaviour?"*

Given the results from the previous chapter, the number of equations necessary to characterize the behaviour of an instruction sequence turns out to be essential. It would, therefore, be desirable to know the relation between the number of instructions of which a program consists, and the maximum number of equations generated when extracting the associated behaviour.
For this number of instructions of a program, we will use the term *length*, in order to distinguish it from the concept of the *norm* of a program. It is crucial to make this distinction, as the number of instructions of a program $X = (u_1; ...; u_n)^\omega$ is n but its norm is infinite.

PGA

Second canonical form

Second canonical form without repetition operator

Since each of the equations in the set expressing the program's behaviour is of form (2.15), (2.16) or (2.17), it seems intuitive that each instruction is translated to at most one equation: only basic instructions and test instructions lead to form (2.15), the termination instruction "!" leads to (2.16), and #0 or jumps to beyond the length of the instruction sequence lead to (2.17). Additionally, if a program without repetition operator ends with a basic instruction, a test instruction, or a jump, the extracted behaviour can contain an extra equation for the deadlock that follows.
Therefore, for any instruction sequence of finite behaviour with n instructions, the number of equations expressing its behaviour cannot exceed $n + 1$.
In practice, the maximum of $n + 1$ will often not be reached, as there are several situations in which fewer equations are needed:

- only one equation is needed to represent deadlock, while several instructions can lead to deadlock
- only one equation is needed to represent succesful termination, while several instructions can lead to succesful termination
- jump instructions not leading to deadlock, are not represented by extra equations

- there can be identical instructions with identical succeeding instructions. These can be represented by the same equation.

However, this maximum can never be *exceeded*. We can prove this with induction:

base case
In second canonical form, the shortest possible instruction sequence contains only one instruction. This can be any of the five primitive instructions, giving rise to three different possible sets of equations:

- a basic instruction or a positive or negative test instruction. Behaviour extraction equation (2.2), (2.3), or (2.4) applies, and thus the extracted behaviour will consist of two equations:

$$\begin{aligned} X_1 &= X_2 \trianglelefteq a \trianglerighteq X_2 \\ X_2 &= D \end{aligned}$$

- a succesful termination symbol. Behaviour extraction equation (2.1) applies. The extracted behaviour will only consist of one equation:

$$X_1 = S$$

- a jump instruction. Behaviour extraction axiom (2.5) applies. The extracted behaviour will only consist of one equation:

$$X_1 = D$$

In this case, the number of instructions $n = 1$, and the maximum number of equations is $2 = n + 1$.

induction step
Any longer instruction sequence can be seen as $u_{n+1}; X$ with $X = u_n; u_{n-1}; ...; u_1$ and u_k, $k \in \mathbb{N}^+$ primitive instructions. Each additional instruction can at most add a single equation to the set. As such, with each additional instruction sequence length n increases with 1, and so does the maximum number of necessary equations $n + 1$.

Second canonical form with repetition operator

For second canonical forms ending with a sequence of instructions to which the repetition operator is applied, the induction of the previous paragraph is slightly different:

base case
The shortest possible instruction sequence in second canonical form with repetition operator contains two instructions: one instruction that is *not* infinitely repeated and one instruction that *is* infinitely repeated. The infinitely repeated instruction can be any of the 5 primitive instructions:

- a basic instruction or a positive or negative test instruction. After every execution of this instruction, the same instruction will be executed again. The extracted behaviour will consist of one equation:

$$X_1 = X_1 \trianglelefteq a \trianglerighteq X_1$$

- a succesful termination symbol. $(!)^\omega$ can be unfolded to $!; (!)^\omega$, so behaviour extraction equation (2.6) applies. The extracted behaviour will only consist of one equation:

$$X_1 = S$$

- a jump instruction. Convention (2.14) applies. The extracted behaviour will only consist of one equation:

$$X_1 = D$$

Including the preceding instruction without repetition, the program can be written as $u_2; X$ with $X = (u_1)^\omega$. Therefore, u_2 adds add most one equation to the set. The maximum number of necessary equations equals the number of instructions: $n = 2$.

induction step
Any longer instruction sequence can be seen as one or more instructions preceding the last instruction as part of the infinitely repeated sequence of instructions, one or more instructions preceding the non-repeating instruction, or both. For each additional instruction u_n, the program can be written as one of the following:

$$X = (u_n; Y); Z^\omega$$
$$X = Y; (u_n; Z)^\omega$$

In both cases, the preceding instruction leads to at most one extra equation. With each additional instruction, program length n increases with 1, and so does the maximum number of necessary equations n.

First canonical form

Programs in first canonical form differ from those in second canonical form in two ways:

- jumps into the repeating sequence, if present, are not minimal
- there can be chained jumps in the instruction sequence

Neither of these two possibilities influence the inductions from the previous sections. Therefore, programs in first canonical form also adhere to the given maximum numbers of necessary equations.

PGA-programs in any form

As mentioned Bergstra and Loots [3], any closed PGA-program can be transformed into an instruction sequence equivalent canonical form. This can be shown by induction, as any PGA-program is of one of three forms:

- $X = u$, with u a primitive instruction. X is already in canonical form.
- $X = X_1; X_2$. By the induction hypothesis, there must be canonical forms U_1 and U_2 such that $X_1 = U_1$ and $X_2 = U_2$. Now, there are two possibilities:
  If U_1 contains no repetition operator, then $U_1; U_2$ is in canonical form.
  If U_1 does contain a repetition operator, U_1 can be written as $U_1 = Y; Z^\omega$.
  Thus, $U_1; U_2 = Y; Z^\omega; U_2$.
  Now, by PGA3:
  $Y; Z^\omega; U_2 = Y; Z^\omega$, which is in canonical form.
- $X = (X_1)^\omega$. By the induction hypothesis, there must be a canonical term $U = X_1$. Again, there are two possibilities:
  U contains no repetition operator. In that case, the canonical form of X is given by $U; U^\omega$.
  If U does contain a repetition operator, so $U = Y; Z^\omega$, then by PGA4:
  $X = (Y; Z^\omega)^\omega = Y; Z^\omega; (Y; Z^\omega)^\omega$
  and by PGA3:
  $X = Y; Z^\omega; (Y; Z^\omega)^\omega = Y; Z^\omega$, which is in canonical form.

The length of the program can only increase in the first case of the third form. However, each instruction added to precede the repeating part of the program is identical to its corresponding

instruction in the repeating part, as are its subsequent instructions. Therefore, behaviour extraction does not generate extra equations for these instructions, there are still at most n instructions necessary to describe the associated behaviour:

$$|\langle X \;=\; u_1; u_2; ...; u_n; (u_1; u_2; ...; u_n)^\omega \rangle| \;=\; |\langle X \;=\; (u_1; u_2; ...; u_n)^\omega \rangle|$$

Conclusion

The maximum number of equations to minimally express the behaviour of a program of length n in PGA, is $n + 1$ for programs without repetition operator.
For programs in PGA of length n which do contain at least one instruction to which the repetition operator is applied, at most n equations are needed.
These maximum numbers of equations are independent of whether these programs are in non-canonical form, first canonical form, or second canonical form.

Higher languages

What does the previous mean for programs in higher languages? For each of these languages, a projection to a lower language is defined. This means it is always possible (if necessary, through multiple consecutive projections to increasingly lower languages) to translate a program in a certain higher language to a program in PGA. Since the length of this projection of the program in PGA depends only on the length of the program and the transformations in the projections, the maximum number of equations necessary to characterize the original program's behaviour can be inferred from its original length and the corresponding language.
It is also possible to define a direct system for thread extraction, as is done by Bergstra and Bethke [2] for the language PGLBbt. From such a system, we can infer directly which number of equations is at most necessary to express the behaviour of a program of a given length.
To illustrate, we show these properties for the higher languages mentioned in Chapter 2: PGLA and PGLB.

PGLA

Projection

Any program in PGLA can be projected directly to a canonical form of PGA, after which the behaviour extraction defined on PGA can be applied. There are two cases to be considered: PGLA-programs without repeat instruction and PGLA-programs with at least one repeat instruction:

- If sequence X contains no repeat instruction, the projection is given by PGLA2PGA(X) $= X$. The number of instructions remains the same, and therefore the number of equations at most necessary to characterize the behaviour associated with a PGLA-program of length n without repeat instruction is equal to the number of equations maximally necessary to characterize the behaviour of a PGA-expression without repetition operator of length n: $n + 1$.
- If sequence X contains at least one repeat instruction, all instructions to the right of the left-most repeat instruction are removed. Subsequently, the projection to PGA of the remaining expression $X \;=\; u_1; ...; u_k; \backslash\backslash \#m$ depends on the relation between k and m:
  - if $k \;>\; m$, then PGLA2PGA(X) $= u_1; ...; u_{k-m}; (u_{k-m+1}; ...; u_k)^\omega$
    In this case, the number of instructions of the equivalent expression in PGA is equal to k. As this concerns a PGA-expression with a repetition operator, the maximally necessary number of equations for a PGLA-program of this form is in this case equal to $k+1$. The original expression $X \;=\; u_1; ...; u_k; \backslash\backslash \#m$ included the repeat instruction and was therefore of length $n \;=\; k + 1$. Therefore, the actual number of maximally necessary equations is given by $n - 1$.

– if $k = m$, then PGLA2PGA$(X) = u_1; ...; u_k; (u_1; ...; u_k)^\omega$
In this case, all instructions are included once as a non-repeating part, and once as a repeating part of the PGA-expression. Both occurrences of u_i with $1 \leq i \leq k$ are the same instruction followed by the same consecutive instructions though. Therefore, at most one equation is necessary to represent the associated behaviour. Thus, the maximum number of necessary equations is given by $k = n - 1$.

– if $k < m$, then PGLA2PGA$(X) = u_1; ...; u_k; (\#0)^{m-k}; (u_1; ...; u_k; (\#0)^{m-k})^\omega$
In this case, all instructions are included once as a non-repeating part and once as a repeating part of the associated PGA-expression, and additionally $m - k$ zero-jumps are included. As these instructions are jump-instructions by definition, they do not lead to additional behaviour. Therefore, the maximum number of necessary equations is given by $k = n - 1$.

Since $n = k + 1$ is never larger than the length of X before the possible removal of the instructions right of the left-most repeat instruction, the given maximum of $n - 1$ for each of these cases is the actual maximum for the original expression as well.

Direct extraction

For the direct extraction of behaviour associated with an instruction sequence in PGLA, Table 4.1 can be used. In general,

$$|X|_{PGLA} = |1, X| \text{ as defined in Table 4.1}$$

with an extra rule depending on the occurrence of the repeat instruction and the value of its counter.
To determine the number of equations resulting from this process, several cases need to be considered:

- If a PGLA-expression contains no repeat instruction, the program is given by $X = u_1; ...; u_n$. Behaviour can be extracted directly by the equations in Table 4.1, with

$$|i, X| = D \text{ if } i > n$$

The smallest possible sequence of this form consists of a single primitive instruction. If this is a successful termination symbol, the extracted behaviour will be given by the final equation in 4.1, and consist of one equation:

$$X_1 = S$$

If it is a jump instruction, the extracted behaviour will also consist of a single equation:

$$X_1 = D$$

Finally, if this single primitive instruction is a basic instruction, a positive test instruction or a negative test instruction or it leads to the set of two equations:

$$\begin{aligned} X_1 &= X_2 \trianglelefteq a \trianglerighteq X_2 \\ X_2 &= D \end{aligned}$$

Any longer instruction sequence of this form can be written as $u_1; X$ with $X = u_2; ...; u_n$, where each preceding instruction generates at most one additional equation. Therefore, the maximum number of necessary equations is $n + 1$.

- If a PGLA-expression contains at least one repeat instruction, its behaviour can be extracted by first removing any instructions to the right side of the left-most repeat instruction to obtain an expression of the form $X = u_1; ...; u_k; \backslash\backslash \#m$. The process of extraction now depends on the relation between k and m:

– $k \geq m$.
Behaviour can be extracted by the equations in Table 4.1, with

$$|i, X| = |i - m, X| \text{ if } i > k$$

and the convention that the extracted behaviour in cases of infinite application of equations not leading to any behaviour, equals D.

– $k < m$.
$X = u_1; ...; u_k; \backslash\backslash \#m$ is first converted to $X = u_1; ...; u_k; (\#0)^{m-k}; \backslash\backslash \#m$.
The corresponding behaviour can be extracted by the equations in Table 4.1, with

$$|i, X| = |i - m, X| \text{ if } i > m$$

and, again, the convention that the extracted behaviour in cases of infinite application of equations not leading to any behaviour equals D.

The shortest instruction sequence of this form consists of a single primitive instruction, followed by a repeat instruction: $u_1; \backslash\backslash \#m$. If $m = 1$, this would generate only one equation:

$$\begin{array}{ll} X_1 = D & \text{if } u_1 \text{ is a jump instruction} \\ X_1 = S & \text{if } u_1 \text{ is a successful termination symbol} \\ X_1 = X_1 \trianglelefteq a \trianglerighteq X_1 & \text{if } u_1 \text{is a basic instruction or test instruction} \end{array}$$

However, if u_1 is a basic instruction or test instruction and $m \neq 1$, *two* equations are needed: one to express the behaviour of instruction u_1, and one to express the deadlock that follows. So in general, the length of the sequence is $n = 2$, and the maximum number of necessary equations equals n.
Any longer instruction sequence can be regarded as $u_1; X$ with $X = u_2; ...; u_{n-1}; \backslash\backslash \#m$, where each preceding instruction generates at most one extra equation according to Table 4.1. Therefore, the maximum number of necessary instructions remains n.

Table 4.1: Equations for extraction of behaviour from PGLA-expressions, where $l \in \mathbb{N}^+$

$\|i, X\| = \|i+1, X\| \trianglelefteq a \trianglerighteq \|i+1, X\|$	if $u_i = a$
$\|i, X\| = \|i+1, X\| \trianglelefteq a \trianglerighteq \|i+2, X\|$	if $u_i = +a$
$\|i, X\| = \|i+2, X\| \trianglelefteq a \trianglerighteq \|i+1, X\|$	if $u_i = -a$
$\|i, X\| = D$	if $u_i = \#0$
$\|i, X\| = \|i+l, X\|$	if $u_i = \#l$
$\|i, X\| = S$	if $u_i = !$

To illustrate, we give two examples:
If $X = -a; \#1; \#1; \backslash\backslash \#2$, then

$$\begin{array}{l} |1, X| = |3, X| \trianglelefteq a \trianglerighteq |2, X| \\ |2, X| = |3, X| \\ |3, X| = |4, X| \\ |4, X| = |2, X| \end{array}$$

and thus we find $|X|_{PGLA} = |1, X| = \langle X_1 \rangle$ with

$$\begin{array}{l} X_1 = X_2 \trianglelefteq a \trianglerighteq X_2 \\ X_2 = D \end{array}$$

.

If $X = -a;\#1;+b;\backslash\backslash\#2$, then

$$\begin{aligned} |1,X| &= |3,X| \trianglelefteq a \trianglerighteq |2,X| \\ |2,X| &= |3,X| \\ |3,X| &= |2,X| \trianglelefteq b \trianglerighteq |3,X| \end{aligned}$$

and thus we find $|X|_{PGLA} = |1,X| = \langle X_1 \rangle$ with

$$\begin{aligned} X_1 &= X_2 \trianglelefteq a \trianglerighteq X_2 \\ X_2 &= X_2 \trianglelefteq b \trianglerighteq X_2 \end{aligned}$$

PGLB

Projection

Any instruction sequence in PGLB can be projected to PGA directly by

$$\text{PGLB2PGA}(u_1;...;u_n) = (\psi_1(u_1);...;\psi_n(u_n);\#0;\#0)^\omega.$$

in which function $\psi_j(u)$ changes only the jump-counters, according to Table 4.2.

Table 4.2: function $\psi_j(u)$ for PGLB2PGA

$\psi_j(\#l) = \#l$	if $j+l \leq n$
$\psi_j(\#l) = \#0$	if $j+l > n$
$\psi_j(\backslash\#l) = \#n+2-l$	if $l < j$
$\psi_j(\backslash\#l) = \#0$	if $l \geq j$
$\psi_j(u) = u$	if otherwise

Of the PGA-expression $(\psi_1(u_1);...;\psi_n(u_n);\#0;\#0)^\omega$, at most n primitive instructions generate behaviour, in addition to which the two zero-jumps added at the end add at most one equation for deadlock. Therefore, the maximum number of equations necessary to characterize the behaviour of an instruction sequence in PGLB of length n equals $n+1$.

Direct extraction

Behaviour can be extracted directly from instruction sequences in PGLB with Table 4.3, such that

$$|X|_{PGLB} = |1,X| \text{ as defined in Table 4.3}$$

with the additional convention that infinitely repeated application of equations not leading to any behaviour, equals D.

The shortest possible sequence is of the form $X = u_1$ where u_1 is a primitive instruction. Similar to in previous sections, this primitive instruction generates one equation if it is a jump instruction or a successful termination symbol. The behaviour extracted with Table 4.3 in case of a basic instruction or test instruction would consist of two equations:

$$\begin{aligned} |1,X| &= |2,X| \trianglelefteq a \trianglerighteq |2,X| && \text{as } u_1 = a \\ |2,X| &= D && \text{as } 2 > k \end{aligned}$$

leading to the recursive definition:

$$\begin{aligned} X_1 &= X_2 \trianglelefteq a \trianglerighteq X_2 \\ X_2 &= D \end{aligned}$$

so the maximum number of equations equals $n + 1$.
Any longer sequence in PGLB can be written as $u_1; X$ with $X = u_2; ...; u_n$. For each preceding instruction, at most one equation is added according to Table 4.3. Therefore, the maximum number of equations needed to express the behaviour of an instruction sequence in PGLB of length n equals $n + 1$.

Table 4.3: Equations for extraction of behaviour from PGLB-expressions, where $l \in \mathbb{N}^+$

$\|i, X\| = D$	if $i > n$
$\|i, X\| = \|i+1, X\| \trianglelefteq a \trianglerighteq \|i+1, X\|$	if $u_i = a$
$\|i, X\| = \|i+1, X\| \trianglelefteq a \trianglerighteq \|i+2, X\|$	if $u_i = +a$
$\|i, X\| = \|i+2, X\| \trianglelefteq a \trianglerighteq \|i+1, X\|$	if $u_i = -a$
$\|i, X\| = \|i-l, X\|$	if $u_i = \backslash\#l$ and $i > l$
$\|i, X\| = D$	if $u_i = \backslash\#l$ and $i \leq l$
$\|i, X\| = D$	if $u_i = \#0$ or $u_i = \backslash\#0$
$\|i, X\| = \|i+l, X\|$	if $u_i = \#l$
$\|i, X\| = S$	if $u_i = !$

CHAPTER 5

Conclusions

In order to fully define a process that is the solution to a recursion variable X_i in a linear recursive specification E, it is only necessary to look at the $(n-1)^{th}$ projection, where n is the total number of equations in E.
In particular this means that, in order to determine behavioural equivalence of any two programs, it is only necessary to take the disjoint union of the linear recursive specifications expressing their respective behaviours, and compare the associated $(n-1)^{th}$ projections, where n is the number of equations in the disjoint union.

Furthermore, the maximum value for the number of equations necessary in a linear recursive specification can be determined directly from the number of instructions constituting the program from which this linear recursive specification was derived.

Practically, the combination of these results proves that for any two programs in PGA or one of its higher languages, including those generating infinite behaviour, behavioural equality can be evaluated in a finite and predeterminable number of steps.

Bibliography

[1] A. Barros and T. Hou. A constructive version of AIP revisited. *Electronic Report PRG0802, Programming Research Group, University of Amsterdam*, 2008.

[2] J.A. Bergstra and I. Bethke. On the contribution of backward jumps to instruction sequence expressiveness. *Theory of Computing Systems*, 50(4):706–720, 2012.

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