Course material

**Www-site** for this course: Blackboard (Bb) at

http://blackboard.ic.uva.nl/

**Literature** (offered at Bb as pdf’s)

- Handouts
- Various papers from JLAP51(2), i.e., the *Journal of Logic and Algebraic Programming*, volume 51, number 2 (2002)

These slides provide a summary of all course material

**FYI = For Your Information**-slides: extra knowledge, not required for the exam.
Overview of the course

The course Concurrency Theory 2017/2018 is about three main topics:

1. Program Algebra (PGA) and Thread Algebra (TA), an algebraic approach to imperative programming (≈ first 2.5 lectures)
2. Process Algebra (ACP), an algebraic approach to communicating processes / behaviors and PSF Toolkit (≈ next 4.5 lectures)
3. 1 + 2 + more theory, subjects: state machines, multi-threading, expressiveness, and concurrent machine models (rest)

Objective

- Learning about algebraic methods in the field of concurrency theory (specification languages, modelling of processes, and distributed algorithms)
- Introduction to TCS’s current research and the track programme Foundations of Computing and Concurrency (VU-UvA-MSc. CS)
Course catalogue: Objectives

The student should be able

1. to reason with equational axioms for instruction sequences,
2. to relate these axioms to a simple semantics for instruction sequences,
3. to apply these axioms to small examples, such as algorithms for simple computations on simple data structures,
4. to reason with equational axioms for sequential and concurrent processes,
5. to relate these axioms to a simple semantics, and
6. to apply these axioms to small examples, for example proving the correctness of a communication protocol.
Position & motivation

This course is a constrained choice course in the track (or, specialization) **Foundations of Computing and Concurrency** (FCC) and its main objective is twofold:

1. to introduce the FCC-student to the field of concurrency theory (and to more advanced courses in this field), and
2. to provide a general introduction to this field suitable for MSc students in related disciplines.

Assessment

Written examination (70%) and practical assignments (30%). Submission of the practical assignments is **required**.
Overview of Lecture 1

Contents

- PGA, a quick introduction
- Threads (or behaviors) and behavior extraction
- Some simple expressiveness results

Slides will be available at Bb under Course Documents.¹

Literature

- Handout 1
- Bergstra and Loots, JLAP51(2):125–156

(Also available at Bb under Course Documents.¹)

¹Or: https://staff.science.uva.nl/a.ponse/CT.html
Program algebra, or PGA for short

PGA is an algebraic approach to sequential-imperative and object based programming.

Main question: What is a program?

Rough answer: a sequence of instructions.

We start with a collection $\Sigma$ of basic instructions $a, b, c, \ldots$
In a later stage we will consider more specific basic instructions, such as $[x := 3x + 1]$. For now we only say

the execution of a basic instruction $a$ invokes a behavior:
the action $a$ followed by some subsequent behavior.

(In these slides, colors are used to distinguish basic instructions and actions.)
PGA’s syntax

PGA has five types of (primitive) instructions:

1. for each $a \in \Sigma$ the basic instruction $a$
2. the termination instruction $!$
3. for each $a \in \Sigma$ the positive test instruction $+a$
4. for each $a \in \Sigma$ the negative test instruction $-a$
5. for each $k \in \mathbb{N}$ the jump instruction $\#k$

PGA-programs are these:

- each primitive instruction is a PGA-program
- if $X$ and $Y$ are PGA-programs, so is their concatenation $X; Y$
- if $X$ is a PGA-program, so is its repetition $X^\omega$
Almost no first intuitions

A PGA-program represents a program object, i.e., a sequence of primitive instructions.

- primitive instructions will also be called PGA-instructions, or shortly instructions if the context is clear
- a program object can be infinite: in this case repetition occurs in each representing PGA-program
- program objects are executed single-pass: each instruction is dropped after execution

We postpone to explain the meaning (execution) of PGA’s primitive instructions and first concentrate on syntax.
First identification:

\[ X; (Y; Z) = (X; Y); Z \]

and henceforth we usually omit brackets in repeated concatenations.

Some examples of PGA-programs:

- \[ a; ! \] (two instructions)
- \[ +a; −b; c \] (three instr’s)
- \[ #2; !; #1; +a; ! \] (five instr’s)
- \[ #2; (b; #1; a)^\omega; ! \] (infinitely many instr’s)
Axioms for instruction sequence congruence

\[(X; Y); Z = X; (Y; Z)\] (PGA1)

\[(X^n)^\omega = X^\omega\] (PGA2, \(n > 0\))

\[X^\omega; Y = X^\omega\] (PGA3)

\[(X; Y)^\omega = X; (Y; X)^\omega\] (PGA4)

with \(X^1 = X\) en \(X^{n+1} = X; X^n\), so there are infinitely many axioms (schematized by PGA2).

PGA1 expresses that brackets are not needed in concatenations (but may occur, to enhance readability). PGA2–4 axiomatize repetition.

*Unfolding:* \(X^\omega = X; X^\omega\)
With the axioms for instruction sequence congruence, which were

\[(X; Y); Z = X; (Y; Z)\]  \quad (PGA1)

\[(X^n) = X^\omega\]  \quad (PGA2,  \ n > 0)

\[X^\omega; Y = X^\omega\]  \quad (PGA3)

\[(X; Y) = X; (Y; X)\]  \quad (PGA4)

we can give a proof of \(X^\omega = X; X^\omega\), i.e., of unfolding:

**Proof:**

\[
X^\omega = (X; X)^\omega \quad (PGA2, \ n = 2)
\]

\[= X; (X; X)^\omega \quad (PGA4)\]

\[= X; X^\omega \quad (PGA2)\]

(Of often we don’t mention axioms in proofs.) □
Instruction sequence congruence

PGA-programs are “the same” if they represent the same program object (sequence of instructions). This can always be proved with the axioms PGA1–4, and is called instruction sequence congruence.

Explicit notation: $X =_{isc} Y$.

Instruction sequence congruence is a congruence indeed (i.e., $=_{isc}$ is preserved under the operations of PGA): if $X =_{isc} Y$, then also

- $X; Z =_{isc} Y; Z$
- $Z; X =_{isc} Z; Y$
- $X^\omega =_{isc} Y^\omega$

We usually write $X = Y$ instead of $X =_{isc} Y$. 
First canonical forms

With the axioms PGA1–4 each PGA-program can be rewritten (preserving instruction sequence congruence) to a form

\[ Y, \quad \text{where } Y \text{ contains no repetition, or} \]

\[ Y; Z^\omega, \quad \text{where } Y \text{ and } Z \text{ do not contain repetition.} \]

Here \( Y \) and \( Y; Z^\omega \) are called first canonical forms.

First canonical forms are not unique, e.g.

\[ #4; (a + b)^\omega = #4; a; (b; a + b; a)^\omega \]

but minimal ones are (minimal in length of the expression).
Conversely, with unfolding \((X^\omega = X; X^\omega)\), PGA1 and the \(\omega\)-rule

\[
X = Y; X \Rightarrow X = Y^\omega
\]

one can derive PGA2-PGA4.

**Proof:**

PGA2, e.g. for \(n = 2\):

\[
X^\omega = X; X^\omega
\]
\[
= X;(X; X^\omega)
\]
\[
= (X;X); X^\omega
\]
\[
\Rightarrow X^\omega = (X;X)^\omega
\]

PGA3: \(X^\omega; Y = (X;X^\omega); Y\)

\[
= X;(X^\omega; Y)
\]
\[
\Rightarrow X^\omega; Y = X^\omega
\]

PGA4: *exercise.* \(\square\)
In thread algebra there are two constants for (non-active) threads:

\[ S \] represents *termination* (the terminated thread)

\[ D \] represents for *inaction* or *deadlock*

Execution of a basic instruction \( a \) yields the *action* \( a \), followed by a reply **true** or **false**. The operators on threads are these:

\[ \_ \triangleleft a \triangleright \_ \] (for each \( a \in \Sigma \)). The *postconditional composition* \( P \triangleleft a \triangleright Q \) represents action \( a \) followed by thread \( P \) if **true** was replied, and \( a \) followed by \( Q \) otherwise,

\[ a \circ \_ \] (for each \( a \in \Sigma \)). The *action prefix* \( a \circ P \) is a short notation for \( P \triangleleft a \triangleright P \) (the reply to \( a \) plays no role).
Notation

Action prefix *binds stronger* than postconditional composition, thus

\[ b \circ S \preceq a \triangleright D = (b \circ S) \preceq a \triangleright D \]

and this thread starts with an \( a \)-action, while

\[ b \circ (S \preceq a \triangleright D) \]

is a different thread (it starts with a \( b \)-action).

Note that actions and PGA’s basic instructions come from the same set \( \Sigma \). It is determined by the context what is meant... (and what color is used).

In the following slides we will visualize threads and discuss their relation with PGA-programs.
Visualizing finite threads

We visualize threads as follows:

\[ b \circ (S \triangleleft c \triangleright D) \triangleleft a \triangleright D : \langle a \rangle \]

\[ [ b ] \quad \text{D} \]

\[ \langle c \rangle \]

\[ \text{S} \quad \text{D} \]

- an action between angular brackets represents postconditional composition:
  - the arrow descending to the left represents the true-case
  - the arrow descending to the right represents the false-case
- an action between square brackets represents action prefix
- the initial state is explicitly mentioned
A first intuition

Given the thread

\[ b \circ (S \triangleleft c \triangleright D) \triangleleft a \triangleright D : \langle a \rangle \]

- the actions \( a \) and \( c \) represent the execution of test instructions: upon \texttt{true} and \texttt{false}, the subsequent behavior is different
- the action \( b \) represents the execution of a basic instruction \( b \), the reply \texttt{true} or \texttt{false} is not used

The action \( a \) will be associated with the execution of a test instruction \( +a \) or \( -a \) (and similar for \( c \))...
Thread extraction (or behavior extraction)

The operator $\ldots$ extracts threads from PGA-programs and is defined thus ($u$ a PGA-instruction):

\[
\begin{align*}
|a| &= a \circ D \\
|+a| &= a \circ D \\
|a| &= a \circ D \\
|!| &= S \\
|#k| &= D \\

|!; X| &= S \\
|a; X| &= a \circ |X| \\
|+a; X| &= |X| \trianglelefteq a \triangleright #2; X \\
|a; X| &= |#2; X| \trianglelefteq a \triangleright |X| \\

|#0; X| &= D \\
|#1; X| &= |X| \\
|#k+2; u| &= D \\
|#k+2; u; X| &= |#k+1; X|
\end{align*}
\]

These 13 equations plus the default rule (slide 23) define the execution of all program objects (also of those not expressible as PGA-programs; future lecture on state machines / services).
Examples of thread extraction

\[ \neg a; \#0; b; +c; ! = \#2; \#0; b; +c; ! \leq a \geq \#0; b; +c; ! \]
\[ = b; +c; ! \leq a \geq D \]
\[ = b \circ |+c; ! \leq a \geq D \]
\[ = b \circ (|! \leq c \geq |\#2; !|) \leq a \geq D \]
\[ = b \circ (S \leq c \geq D) \leq a \geq D \]

Thus, \( \neg a; \#0; b; +c; ! \) specifies the thread

\[ b \circ (S \leq c \geq D) \leq a \geq D : \quad \langle a \rangle \]

\[ \downarrow \]

[ b ] \quad D

\[ \text{S} \quad \text{D} \]

The same thread is also specified by (and by many more PGA-programs)

\[ +a; \#2; \#0; +b; \#1; +c; ! \]
Equations for infinite program objects

With unfolding we find \(|(a; + b)\omega| = |a; + b; (a; + b)\omega|\), and hence

\[ |(a; + b)\omega| = a \circ |+ b; (a; + b)\omega|, \]
\[ |+ b; (a; + b)\omega| = |(a; + b)\omega| \trianglelefteq b \triangleright |+ b; (a; + b)\omega|. \]

Shorter and easier to read: \(P = |(a; + b)\omega|\) with

\[ P = a \circ Q, \]
\[ Q = P \trianglelefteq b \triangleright Q. \]

(Also the identifier \(Q\) is arbitrarily chosen.) In this case, \(P\) in the above (system of) two equations defines/specifies the thread \(|(a; + b)\omega|\). In a picture:

\[ P : \begin{array}{c}
\begin{array}{c}
\downarrow \quad [a] \\
\langle b \rangle
\end{array}
\end{array} \]
Default rule. If the 13 equations for thread extraction do not generate any behavior, as in

\[ |(#2)^{\omega} | = | #2; #2; (#2)^{\omega} | = | #1; (#2)^{\omega} | = |(#2)^{\omega} | , \]

we assign D.

Property: \[ |X| = |X; (#k)^{\omega} | \] (convince yourself).

Definition: two programs \(X\) and \(Y\) are behavioral equivalent if \( |X| = |Y| \). Notation: \( X \equiv_{be} Y \).

Fact: behavioral equivalence is not a congruence.

E.g., \#0 \( \equiv_{be} \) \#1 (because D = D)
\#0;! \( \not\equiv_{be} \) \#1;! (because D \( \not\equiv \) S).
Some expressiveness issues

In PGA, basic and negative test instructions are redundant.

Idea: First,  \( |X| = |X; (#0)\omega| = |X; #0; \ldots; #0; (#0)\omega| \).

Secondly,  \( |a; u; Y| = |a; #1; u; Y| \),
\( |\neg a; u; Y| = |\neg a; #2; u; Y| \).

Example:  \( |\neg a; !| = |a; #2; !; (#0)\omega| \)  \((= |a; #2; !; #0|)\).

Even stronger: iterative canonical forms \((u_1; \ldots; u_k)^\omega\) with \(u_i\) a positive test, a jump or a termination instruction are sufficient to express each PGA-behavior.

(Lecture 2: each PGA-behavior can be programmed by an iterative canonical form.)
Expressiveness results continued

In PGA, small jumps are redundant.

**Proof:** Use iterative canonical forms \((u_1; \ldots; u_k)\omega\) and increment all jump counters with \(k\). □

Example:

\[
\left| (+a; +b; \#2)^\omega \right| = \left| (+a; +b; \#5)^\omega \right| \\
= \ldots \\
= \left| (+a; +b; \#3002)^\omega \right|
\]
FYI - In PGA, large jumps are necessary ($\Sigma \neq \emptyset$)

Proof (sketch): Given some $n > 2$, we define a thread that can NOT be programmed without jumps larger than $n$: for $j = 1, ..., n + 1$ let

$$Y_j = a_j^i; +a; !; \#m_j$$

with all $m_j$ such that $Y_j$ repeats itself in the repetition part of

$$X = +a; \#[to Y_1]; ...; +a; \#[to Y_{n+1}]; !; (Y_1; ...; Y_{n+1})^\omega$$

(L1-picture.pdf).

Assume the behavior of $X$ can be defined with jumps of at most $n$, then also by a first canonical form

$$u_1; ...; u_m; (v_1; ...; v_k)^\omega$$

Between each adjacent pair of $Y_1$-instructions in $(v_1, ..., v_k)^\omega$ there are at most $n - 1$ other instructions, but these represent the $n$ different threads defined by the repetition of $Y_2, ..., Y_{n+1}$: contradiction. □
In this lecture we introduced many concepts

- PGA, a quick introduction
- TA, threads (or behaviors), and behavior extraction
- Some simple expressiveness results

Take your time for practice and study. E.g., draw from slide 20 the conclusion that the meaning of a positive test instruction $+a$ in a PGA-program is to execute the action $a$, and if the reply is

- true, then to execute the next instruction if present and otherwise deadlock
- false, then to skip the next instruction and execute the instruction thereafter if present, and otherwise deadlock