Execution Architectures for Program Algebra

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

We investigate the notion of an execution architecture in the setting of the program algebra PGA, and distinguish two sorts of these: *analytic* architectures, designed for the purpose of explanation and provided with a process-algebraic, compositional semantics, and *synthetic* architectures, focusing on how a program may be a physical part of an execution architecture. Then we discuss in detail the Turing machine, a well-known example of an analytic architecture. The logical core of the halting problem — the inability to forecast termination behavior of programs — leads us to a few approaches and examples on related issues: *forecasters* and *rational agents*. In particular, we consider architectures suitable to run a Newcomb paradox system and the Prisoner's Dilemma.

Key words: Halting problem, Execution of programs, Program algebra, Turing machine.

Contents

2 June 2004

1 Introduction

The program algebra PGA as introduced in [6] aims at the clarification of the concept of a program at the simplest possible level. Having available a rigid definition of what a program is, the subject of how programs may be used raises compelling questions. This paper focuses on the notion of an execution architecture. This notion is more general than that of a machine and admits many different forms of interaction between a program and its context.

First, an attempt is made to cover the most important phenomena regarding programs in a context. As programs are modeled semantically, and independently of any execution environment, by means of polarized processes it is unavoidable to contemplate computable polarized processes as a general semantic category. It turns out that computable instruction streams can describe all computable processes. Using finite programs only (regular instruction streams), taking an execution architecture with the well-known Turing Machine Tape (TMT) as a reactor is sufficiently powerful to denote all computable polarized processes as well.

Next, two kinds of architectures are defined: analytic architectures (AnArch) and synthetic architectures (SynArch). An AnArch serves to model how a program may

be executed in a context, by making explicit its interaction with other system components, in particular the so-called reactors. A SynArch focuses on how a program may be a physical part of a context. The AnArch is useful for explanation, while a SynArch may play a role during construction. It is shown that all SynArch's admit a specification in a (non-polarized) process algebra with abstraction and recursion operators.

Then some special analytic execution architectures are discussed in detail in order to cover a range of fundamental phenomena each having to do with programs under execution in an environment.

An enhanced version of the TMT is developed in the form of a reactor ETMT. More specifically, ETMT is a state machine, i.e., it has no interaction with other parties than the control. Finite control is phrased in terms of programs executed in an analytical environment providing only the ETMT. In this setting the halting problem takes the form of the nonexistence of certain programs, which is demonstrated in full detail.

Ignoring the (E)TMT, the halting problem reduces to its logical core: the inability to forecast termination behavior of programs that may use the results of forecasting. It is shown how an analytic architecture can be used to give a sound definition of a forecasting reactor, and it is demonstrated that a correct forecaster needs to escape from the two classical truth values. This brings the halting problem close to some logical paradoxes, in particular the liar paradox.

A forecasting reactor intends to provide replies that correspond to (future) facts. A rational agent reactor has the objective to achieve certain goals by giving appropriate replies for specific requests. It is shown that again in some cases a rational agent needs to use more truth values than true and false.

Combining rational agent reactors and forecasting reactors, one obtains a remarkable setting. This is a setting that admits the famous Newcomb paradox [13]. This paradox seems to prove that the very concept of a forecaster reliably forecasting a rational agent is utterly problematic. Nevertheless this is done all the time in chess games, stock market transactions, war gaming and so on. Using the analytic architectures and some exotic process algebra involving the constant 0 from [2], a formalization of one reactor forecasting another reactor is given. The Newcomb paradox now shows up as follows: given a fixed execution architecture (viewed as a geometric structure with several components), its process semantics determines what a rational agent reactor should best reply in order to achieve a specific objective. The normal process semantics predicts one reply as being rational, whereas the semantics specifically tailored to forecasting predicts a different reply. But the normal semantics is so robust that it seems to take into account the possibility that one reactor predicts the behavior of another reactor just as well. The novelty of this section may lead in the very presence of a precise formalization of the conditions

required to run both executions of the 'Newcomb paradox system.' As a last and related example, we model the well-known Prisoner's Dilemma.

A related subject in the context of this paper is the undecidability of virus detection as described by Cohen [12]. In the setting of program algebra one can consider execution architectures that take security matters into account, and establish an analysis in similar style; we plan this as future research.

The further content of this paper is divided into four parts: in Section 2 we recall some program algebra. In the next section we introduce execution architectures. Then, in Section 4, we study the Turing machine as an example of an analytic architecture. Finally, in Section 5, we focus on forecasting reactors and rational agents in the setting of (analytic) execution architectures.

2 Basics

In this section we recall some program algebra (PGA) and its relation with basic polarized process algebra. Then we introduce some program notations based on PGA. Finally, we show that computable, polarized processes can be expressed.

2.1 PGA, some Basics of Program Algebra

Program Algebra (PGA, [6]) provides a systematic setting for the study of sequential, imperative programming. In this paper we will use PGA as a vehicle to study fundamentals of program execution in the context of self-referencing programs. In this section we discuss the syntax and semantics of PGA, and some program notations based on PGA.

Given a set of Σ of basic instructions, the syntax of PGA (PGA_Σ) is based on the following primitive instructions:

Basic instruction $a \in \Sigma$. 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 $a \in \Sigma$. 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 instruc-

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

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

Table 1.
Axioms for PGA's instruction sequence congruence

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

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

So-called *structural equivalence* is obtained by abstracting from chained jumps. For PGA-expressions it is axiomatized by the axioms PGA1-8 ($n, m, k \in \mathbb{N}$) in Tables 1 and 2.

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

Table 2.
Additional axioms for PGA's structural congruence

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. Moreover, using PGA1-8, each canonical form can be uniquely minimized in terms of the occurring jump counters and number of instructions. For example, the minimal canonical form for $+a;\#2;(-b;\#2;-c;\#2)^\omega$ is $+a;(\#0;-b;\#0;-c)^\omega$.

We shall use the abbreviation SPI for Sequence of Primitive Instructions. A SPI is also called a *program object*, or sometimes shortly, a *program*.

Given Σ, now considered as a set of *actions*, behavior is specified in BPPA_Σ by means of the following constants and operations:

Termination. The constant S represents (successful) termination.

Inaction, Deadlock or *Divergence.* The constant D represents the situation in which no subsequent behavior is possible.

Post conditional composition. For each action $a \in \Sigma$ and behavioral expressions P and Q in BPPA_Σ,

$$P \trianglelefteq a \trianglerighteq Q$$

describes the behavior that first executes action a, and continues with P if `true` was generated, and Q otherwise.

Action prefix. For $a \in \Sigma$ and behavioral expression $P \in \text{BPPA}_\Sigma$,

$$a \circ P$$

describes the behavior that first executes a and then continues with P. Action prefix is a special case of post conditional composition: $a \circ P = P \trianglelefteq a \trianglerighteq P$.

The *behavior extraction* operator $|X|$ assigns a behavior to program object X. Structural equivalent objects have the same behavior.

Behavior extraction is defined by the thirteen equations in Table 3, where $a \in \Sigma$ and u is a primitive instruction:

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

Table 3.
Equations for behavior extraction on PGA

Some examples: $|(\#0)^\omega| = |\#0; (\#0)^\omega| = D$ and

$$\begin{aligned} |{-}a; b; c| &= |\#2; b; c| \trianglelefteq a \trianglerighteq |b; c| \\ &= |\#1; c| \trianglelefteq a \trianglerighteq b \circ |c| \\ &= |c| \trianglelefteq a \trianglerighteq b \circ c \circ D \\ &= c \circ D \trianglelefteq a \trianglerighteq b \circ c \circ D. \end{aligned}$$

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

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

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

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

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

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

and therefore, $|a^\omega| = a \circ |a^\omega|$

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

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

$$\vdots$$

In such cases the behavior of X is infinite, and can be represented by a finite number of behavioral equations, e.g., $|(a;+b;\#3;-b;\#4)^\omega| = P$ and

$$P = a \circ (P \trianglelefteq b \trianglerighteq Q),$$

$$Q = P \trianglelefteq b \trianglerighteq Q.$$

More precisely, a polarized behavior P is called *regular* (over Σ) if it can be characterized by a finite number of equations in the following way: $P = E_1$ and for $i = 1, ..., n$, $E_i = t_i$ with t_i is either D, S or $E_j \trianglelefteq a \trianglerighteq E_k$ for some j, k in $1, ..., n$. Now any PGA-program defines a polarized regular behavior, and conversely, each regular polarized behavior can be described in PGA.

Note 1 Observe that the following identity holds: $|X| = |X;(\#0)^\omega|$. This identity characterizes that for a finite program object, a missing termination instruction yields inaction. Conversely, this identity makes six out of the thirteen equations in Table 3 derivable (namely, those for programs of length 1 and the equation $|\#k+2;u| = D$).

2.3 The program notations PGLA, PGLB, PGLC with FMN

PGLA is a programming language based on PGA: the only construct (operation) is concatenation, and instead of the repeat operator $(\_)^\omega$ PGLA contains the *repeat instruction* $\backslash\backslash k$ for any $k > 0$, which upon execution repeats the k instructions

that are to the left of it. If there are not that many instructions, the leftmost sequence is padded with $\#0$-instructions, e.g., $+a;\backslash\backslash 2$ behaves as $+a;\#0;\backslash\backslash 2$ or as $+a;\#0;\backslash\backslash 2;b$, and thus as $|(+a;\#0)^\omega|$. We write

$$|X|_{pgla}$$

for the behavior of PGLA-program X. This is defined by a projection function `pgla2pga` from PGLA-programs to PGA-expressions: $|X|_{pgla} = |\texttt{pgla2pga}(X)|$ (see [6]).

The language PGLB is obtained from PGLA by adding backwards jumps $\backslash\#k$ and leaving out the repeat instructions $\backslash\backslash k$. For example, $+a;\#0;\backslash\#2$ behaves as $+a;\#0;\backslash\backslash 2$. However, $+a;\#2;\backslash\#2;b$ behaves not like $+a;\#2;\backslash\backslash 2;b$, in the case that action a generates true, it jumps *over* the backward jump $\backslash\#2$ and performs b, in symbols: $|+a;\#2;\backslash\#2;b|_{pglb} = P$ with $P = b \circ D \trianglelefteq a \trianglerighteq P$. This is defined with help of a projection function `pglb2pgla` by $|X|_{pglb} = |\texttt{pglb2pgla}(X)|_{pgla}$.

PGLC is the variant of PGLB in which termination is modeled implicitly: a program terminates after the last instruction has been executed and that instruction was no jump into the program, or after a jump outside the program. The termination instruction ! is not present in PGLC. For example, $|+a;\#2;\backslash\#2;b|_{pglc} = P$ with $P = b \circ S \trianglelefteq a \trianglerighteq P$.

FMN basic instructions may either have a focus or not. If no focus is present an execution architecture will use a default focus instead. A focus represents a part of a system able to process a basic instruction and to respond subsequently with a boolean value. Such a part may e.g. be called a reactor, a coprogram or an instruction execution agent. The second part of an instruction with focus (and the only part of an instruction without focus) consists of a method. Focus and method are combined by means of a '`.`'. Focus and method may both consist of alphanumeric ASCII (see [1]) sequences, starting with a letter from the alphabet and allowing a colon (`:`) as a separator of parts. Here are some possible typewritten instructions:

```
registers:3.assign:x:to:y
stack:3.push:5
stack:17.pop
table:2.insert:5:at:2
```

A formal CF grammar of FMN is omitted. If basic instructions are taken from FMN and programs are given in PGLA, the resulting notation is termed PGLA:FMN (or pgla:fmn). Here is a PGLB:FMN program:

```
ba2;Bb.de:true;\#1;-a:2.b:3;#5;A:true.false:5.
```

2.4 Computable Polarized Processes

A polarized process is *computable* if it can be represented by an identifier E_1 and two computable functions g, f in the following way ($k \in \mathbb{N}$):

$$E_k = \begin{cases} D & \text{if } g(k) = 0, \\ S & \text{if } g(k) = 1, \\ E_{\langle k+f(k),0\rangle} \trianglelefteq a_{g(k)} \trianglerighteq E_{\langle k+f(k),1\rangle} & \text{if } g(k) > 1. \end{cases}$$

Here we use the bijective pairing function $\langle \_, \_ \rangle$ defined by $\langle n, m\rangle = \frac{1}{2}((n+m)^2 + 3m + n)$. So $\langle n, 0\rangle > n < \langle n, 1\rangle$ if $n > 0$.

Theorem 1 *PGA instruction sequences are universal: for each computable polarized process α there is an instruction sequence with α as its behavior.*

Proof. Let E_1 be a computable polarized behavior as defined above. Then we define

$$\tilde{E}_k = \begin{cases} \#0; \#0; \#0 & \text{if } g(k) = 0, \\ !; !; ! & \text{if } g(k) = 1, \\ +a_{g(k)}; \#3 \cdot \langle k + f(k), 0\rangle - 3; & \\ \qquad \#3 \cdot \langle k + f(k), 1\rangle - 4 & \text{if } g(k) > 1. \end{cases}$$

It is easily seen that $E_1 = |\tilde{E}_1; \tilde{E}_2; ...|$ (or $E_k = |\tilde{E}_k; \tilde{E}_{k+1}; ...|$). □

Furthermore, PGA's repeating sequences of instructions are universal with the aid of a state machine TMT if we restrict to a finite number of actions:

Theorem 2 *For each computable polarized process α there is a closed PGA-term X such that $|X|/_{\mathrm{tmt}}\mathrm{TMT} = \alpha$.*

Here the notation $P/_f S$ stems from [10] and defines the interaction between a behavior P and a so-called state machine or reactor S via focus (channel) f. State machines are used to support program control, and will be further dealt with in Section 3.3. The theorem above is a standard result in the setting of Turing machines (see, e.g., [15,14]), given the fact that finite control can be modeled in PGA.

3 Execution Architectures

In this section we focus on programs in an execution architecture. We will use ACP-based process algebra to model so-called 'analytical architectures'. Finally, we try to clarify the role of programs (binaries) in machines.

3.1 Analytic versus Synthetic Architectures

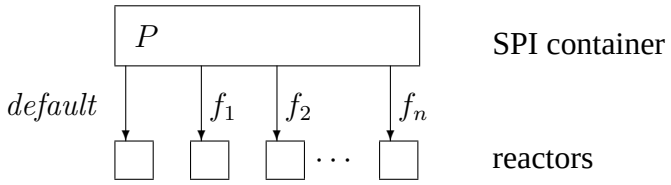
We consider the following types of architectures:

Analytic Architecture (AnArch): provides a hypothetical decomposition of a system into parts. An AnArch can serve as an explanation of a setting in a black box context (the system is seen as a blackbox, with the AnArch describing its internals for the sake of explanation). An AnArch will not be on the pathway to construction.

Synthetic Architecture (SynArch): an architecture (description of how a whole is composed from parts) providing information on the true (or proposed, intended) internal structure of a system.

Often a compositional semantic technique is absent. Parts are organs, the role of which may be investigated later on.

The proposed execution architecture for PGA is an AnArch. For instance a component is provided containing an instruction sequence, able to deliver one instruction at a time. No attention is paid to the way in which a SPI may in fact be stored or generated. We call such a component a SPI container, and visualize an AnArch in the following way:



Here P represents a SPI using FMN-notation.

Each of the reactors may engage in external communications. The channels $default$, f_1, ..., f_n play a reserved role and are supposed not to be composed with other parts of the AnArch or any extension of it.

A reactor R is unaware of its name. It uses actions $r_{serv}(a)$, and $s_{serv}(\mathsf{true})$ and $s_{serv}(\mathsf{false})$ for communication with the SPI container. Reactors are assumed to

satisfy the following requirement: if α is a trace of R then

$$\alpha \cap A_{service} = (\{r_{serv}(a) \mid a \in \Sigma\} \cdot \{s_{serv}(\mathsf{true}), s_{serv}(\mathsf{false})\})^*.$$

When plugged into the AnArch at focus f_i, *serv* $\mapsto$ f_i will be renamed in the actions.

3.2 Processes and Programs

A process is a mathematical entity in a space of processes (like a number being an element of a field). The design of the process space depends on the underlying theory of processes used. We will use ACP ([5], for a recent explanation see [8,11]), but many other process theories can be used instead. In this section, we shall shortly recall some ACP.

The purpose of the use of processes is *specification*. Here, 'specification' is used in a fairly limited way: it must be compared with 'quantification' (stating numerical sizes) and 'qualification' (expressing objectives, goals, methods and reasons). Furthermore, specification stands for the *specification of behavior*. Specification need not be perfect (i.e., it may provide an approximation of a system rather than a perfect view, be it at some level of abstraction). Specification has no a priori place in some artifact construction life cycle, just as quantification or qualification.

A process expression, e.g.

$$r_a(b)(s_a(\mathsf{true})r_a(c) + s_a(\mathsf{false})\delta)$$

provides a text that represents a process (that is, a specification of behavior), namely the process that first performs action $r_a(b)$ (receive along channel a the value b) and then chooses to perform either $s_a(\mathsf{true})r_a(c)$ (send value true along channel a and then perform action $r_a(c)$) or $s_a(\mathsf{false})\delta$. In a picture (where $P \xrightarrow{a} Q$ denotes that P evolves into Q by performing action a):

$$\begin{array}{ccc}
r_a(b)(s_a(\mathsf{true})r_a(c) + s_a(\mathsf{false})\delta) & & \\
\downarrow r_a(b) & & \\
s_a(\mathsf{true})r_a(c) + s_a(\mathsf{false})\delta & & \\
s_a(\mathsf{true}) \swarrow & & \searrow s_a(\mathsf{false}) \\
r_a(c) & & \delta \\
r_a(c) \downarrow & & \\
\surd & &
\end{array}$$

Here δ is the symbol that stands for *inaction* or *deadlock*, and can be compared with the constant D in polarized process algebra, and $\surd$ denotes (successful) termination.

In a similar way, a program expression

$$a.b{:}7; (+c; \#4, -e.f)^\omega$$

represents a SPI. However, there is a crucial difference: suppose process expression X denotes a specification of system S, say $X = [[S]]$, or at least, X is a reasonable approximation of S. Now it is *not* plausible to expect that X or any form of X constitutes a part of S in any SynArch for S. On the other hand, if S is a system executing program p denoted with program expression p, then it is plausible that a SynArch of S contains, perhaps in a transformed (compiled) form, p as a part.

Process expressions occur as parts of systems that analyze or simulate other systems. The following AnArch is perfectly acceptable:

$$S \boxed{P}$$

S contains process expression P and behaves as P, thus S is a P-simulator. As a SynArch this makes little sense. Moreover, simulation is only one of many objectives supported by processes. Calculation and verification is another and probably more important one.

We end this section by recalling some ACP. The signature of ACP has a constant δ and constants for actions. Furthermore, ACP has binary operators $+$ (alternative composition), $\cdot$ (sequential composition), $\parallel$ (parallel composition, merge), $\lfloor\!\lfloor$ (left merge), and $\mid$ (communication merge). Finally, there is a unary renaming operator ∂_H (encapsulation) for every set H of actions, which renames the actions in H into δ. We use infix notation for all binary operators, and adopt the binding convention that $+$ binds weakest and $\cdot$ binds strongest. We suppress $\cdot$, writing xy for $x \cdot y$.

Parallel composition in ACP satisfies the law

$$x \parallel y = (x \lfloor\!\lfloor y + y \lfloor\!\lfloor x) + x \mid y,$$

where $\lfloor\!\lfloor$ is as $\parallel$ with the restriction that the first action must be one from the left argument, while $\mid$ has the restriction that the first action must be a communication.

Communication in ACP is predefined on the set of actions. For example, $a|b = c$ implies $a \parallel b = (ab+ba)+c$. Encapsulation can be used to enforce communication between different parallel components, e.g., $\partial_{\{a,b\}}(a \parallel b) = (\delta\delta + \delta\delta) + c = c$ (by various laws for $\lfloor\!\lfloor$ and δ, such as $x + \delta = x$ and $\delta x = \delta$).

3.3 Compositional Process Specification

We use the notation $[[P]]$ for process semantics of a polarized behavior P in a symmetric, concrete (i.e., without the silent step τ) process algebra:

$$[[S]] = t,$$
$$[[D]] = t^*\delta,$$
$$[[P \trianglelefteq a \trianglerighteq Q]] = s_{default}(a)(r_{default}(\mathsf{true})[[P]] + r_{default}(\mathsf{false})[[Q]]),$$
$$[[P \trianglelefteq f.a \trianglerighteq Q]] = s_f(a)(r_f(\mathsf{true})[[P]] + r_f(\mathsf{false})[[Q]]).$$

Here x^*y is defined by the law $x^*y = x(x^*y) + y$ (see [3]). Taking δ for y and using the ACP-axiom $x + \delta = x$, it follows that $t^*\delta$ behaves as t^ω, i.e., an infinite sequence of t-actions.

For each of the channels $default, f_1, ..., f_n$, the following communications are defined:

$$r_f(a)|s_f(a) = c_f(a) \text{ for } a \in \Sigma \cup \{\mathsf{true}, \mathsf{false}\}.$$

(Recall that Σ is the set of basic PGA-instructions).

Given a polarized behavior P and reactors $R_0, ..., R_n$, we define a concrete analytical architecture, notation cpgaEA, and an abstract one, pgaEA:

$$\text{cpgaEA}(P,\ default{:}R_0, f_1{:}R_1, ..., f_n{:}R_n) =$$
$$\partial_H([[P]] \between \rho_{serv \mapsto default}([[R_0]]) \between \rho_{serv \mapsto f_1}([[R_1]]) \between ... \between \rho_{serv \mapsto f_n}([[R_n]]))$$

with encapsulation set $H = \{r_i, s_i \mid i = default, f_1, ..., f_n\}$. Here, the encapsulation enforces communication between the different parallel components. Furthermore, the renaming operator $\rho_{serv \mapsto f}$ renames the channel name $serv$ to f.

Furthermore,

$$\text{pgaEA}(P,\ default{:}R_0, f_1{:}R_1, ..., f_n{:}R_n) =$$
$$\tau_I(\text{cpgaEA}(P, default{:}R_0, f_1{:}R_1, ..., f_n{:}R_n))$$

with abstraction set $I = \{t, c_{default}, c_{f_1}, ..., c_{f_n}\}$: all actions in set I are renamed to τ, the silent step that satisfies the axiom $x\tau = x$. In common process semantics, $\tau^*\delta = \tau\delta$ (cf. [9]).

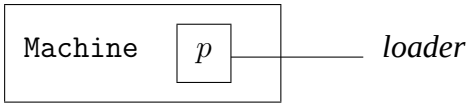
Now cpgaEA$(P, \mathit{default}{:}R_0, f_1{:}R_1, ..., f_n{:}R_n)$ and its pgaEA-variant are computable if P and all R_j are.

A reactor R is called a *state machine* if R has only actions $r_{serv}(a)$, $s_{serv}(\mathsf{true})$ and $s_{serv}(\mathsf{false})$, i.e., no external events, only update of its memory state and computation of boolean output. Based on [10] (describing the interaction between a program or behavior and a state machine), we can prove the following result:

Theorem 3 *Let $\overline{R} = R_0, R_1, ..., R_k, R_{k+1}, ..., R_n$ with $R_{k+1}, ..., R_n$ state machines. Then pgaEA$(P,\ \overline{f_i{:}R_i}) =$ pgaEA$(P/_{f_{k+1}} R_{k+1} .../_{f_n} R_n,\ R_0, ...R_k)$.*

3.4 Synthetic Architectures for Binaries

In this section we try to clarify the role of programs in machines. A binary is just a finite $\{0,1\}$-sequence (i.e., a binary file). Consider the following SynArch:



This SynArch displays a machine `Machine` containing binary p as a part. It has a special port named *loader* used to enter p in `Machine` bitwise.

Assuming that `Machine` is a classical piece of computing machinery, a specification $[[M(p)]]$ for the behavior of p will be a computable process (see [8]). `Machine` can be specified as follows:

$$\begin{aligned}
[[\texttt{Machine}]] &= M_{loading}(\epsilon),\\
M_{loading}(\sigma) &= r_{loader}(0) \cdot M_{loading}(\sigma 0) +\\
&\quad\ r_{loader}(1) \cdot M_{loading}(\sigma 1) +\\
&\quad\ r_{loader}(\mathit{eof}) \cdot [[M(p)]]
\end{aligned}$$

with *eof* an end-of-file marker.

It is reasonable to expect that $[[M(p)]]$ depends uniformly (in the sense of computability theory, see e.g., [15]) on p. Then, also $[[\texttt{Machine}]]$ itself is a computable process:

Theorem 4 *The process $[[\texttt{Machine}]]$ can be denoted modulo branching bisimulation equivalence in ACP extended with τ, $*$, \$, and finitely many auxiliary actions.*

This result is proven in detail in [8]. The operation \$ (called *push-down*) is defined by $x\$y = x(x\$y)(x\$y) + y$, the only τ-law used is $x\tau = x$.

For appropriate encapsulation set H and abstraction set I, we find:

$$\tau \cdot [[M(p)]] = \tau_I \circ \partial_H(S(p) \parallel [[\texttt{Machine}]])$$

where $S(p) = s_{loader}(p_0) \cdot ... \cdot s_{loader}(p_n) \cdot s_{loader}(\textit{eof})$.

Let `pgla:fmn2bin4m` (where `bin4m` abbreviates "binaries for `Machine`") be a mapping from PGLA:FMN to bit sequences. Then `pgla:fmn2bin4m` is a code generator mapping if the following holds for all $X \in$ PGLA:FMN:

$$\text{pgaEA}(|X|_{pgla:fmn},\ \overline{f_i{:}R_i}) = [[M(\texttt{pgla:fmn2bin4m}(X))]].$$

That is: the analytic architecture pgaEA (with its set of reactors) *explains* (i.e., corresponds to) the synthetic architecture SynArch M. In practice one is happy if this works for all X with a size of less than k Mb (for some k).

The following jargon is useful:

(1) PGLA:FMN - middle code or intermediate code.
(2) A machine (program) producing $\texttt{pgla:fmn2bin4m}(X)$ from X is a code generator (or compiler back end) for `Machine`.
(3) The concept of a 'machine code' can not be defined here: clearly, some p are more useful than other p's. But there is no obvious criterion regarding $[[M(p)]]$ to select the binaries for `Machine` from arbitrary bit sequences.
(4) A higher program notation, say PGLX, can be understood if a projection

    `pglx2pgla:fmn`

    to PGLA is known and a pgaEA such that

    $$[[X]] \stackrel{\text{def}}{=} \text{pgaEA}(|\texttt{pglx2pgla:fmn}(X)|_{pgla:fmn},\ \overline{f_i{:}R_i})$$

    corresponds to the intended meaning of program X. A *compiler* is a system (or a program for a system) that allows to compute `pglx2pgla:fmn` (or an optimized version of it that produces semantically equivalent behavior).

For a PGLX-expression X, we then find

$$[[X]] = [[M(\texttt{pgla:fmn2bin4m}(\texttt{pglx2pgla:fmn}(X)))]]$$

and it is common practice to call $\texttt{pgla:fmn2bin4m}(\texttt{pglx2pgla:fmn}(X))$ a *program*.

This is one of the possible justifications for the qualification of a binary that is part of a SynArch as a program. To fix the nature of this qualification, its kind is qualified as follows:

Code generator mapping range criterion: a binary p is a program if it is in the range of a code generator mapping (in a setting that explains the behavior of $M(p)$ via an AnArch).

The qualification of p as a program via the code generator projection mapping criterion seems to be at odds with the basis of PGA because PGA starts from the assumption that *a program is a sequence of instructions* (see [6]). However, if `pgla:fmn2bin4m` is computable, it has a semi-computable inverse, say

$$\texttt{bin4m2pgla:fmn}$$

and p qualifies as a program because of the projection semantics:

$$|p|_{bin:m} = |\texttt{bin4m2pgla:fmn}(p)|_{pgla:fmn}.$$

Of course, it is immaterial that `pgla:fmn2bin4m` is *taken* to be an inverse of `pgla:fmn2bin4m`. What matters is: for all (or as many as one cares) p,

$$\begin{aligned}[[M(p)]] &\stackrel{\text{def}}{=} \text{pgaEA}(|\texttt{bin4m2pgla:fmn}(p)|_{pgla:fmn},\ \overline{f_i{:}R_i}) \\ &(= [[M(\texttt{pgla:fmn2bin4m}(\texttt{bin4m2pgla:fmn}(p)))]]).\end{aligned}$$

Thus, the code generator mapping criterion is consistent with the PGA-criterion for being a program.

Note 2

1. Having a far more detailed SynArch at hand with p as a part, one may find other justifications for qualifying p as a program. However, we failed to develop such a story with any form of acceptable generality.

2. The projection `bin4m2pgla:fmn` may be called a disassembler-projection (ignoring the complexity of loading). Then, if the qualification of p as a program in $M(p)$ is justified by means of the code generator mapping criterion, a disassembler-projection semantics of p is (implicitly) known/given.

3. The justification of the qualification of p in $M(p)$ (p as a part of the SynArch $M(p)$) is itself an argument of a certain form: *qualification on the basis of a most plausible history*. (If we see an object when it is a dead body, of course we see it if it was a living individual of some species that subsequently died. How else could the object have come into existence? If we see p in `Machine` where $p =$ `pgla:fmn2bin4m`(X), that must be related to p's history. How else would it have originated? I.e., p is just another form or phase of X, like a dead body being another phase of a living body.)

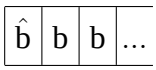
4. The middle code exists at the instruction sequence level (in PGA:FMN or its machine readable version PGLA:FMN). It is at the same time *target code for projection semantics*. Given a SynArch $M(...)$, its binaries are also called *object code*.

4 An Analytic Architecture for Turing Machines

In this section we consider an enhanced version of the Turing machine Tape, and a PGA-based language for programming it. We prove the unsolvability of the halting problem, and show that this problem becomes decidable if we restrict our language sufficiently.

4.1 The Turing machine

The original reference to the Turing machine is [16]. A Turing machine M consists of a finite control, a tape, often visualized in the following style:



where b stands for "blank" (i.e., a blank square), and a head that can be used for reading and writing on that tape. Usually, the tape has a left end, and extends indefinitely to the right. The head can never fall off the tape (at the left side). The control is such that it distinguishes a halting state, after which control is terminated and the tape can be inspected for possible output. In a non-halting state, control prescribes some action to be undertaken and the next control state. Actions are either: "write a symbol in the square" (where write a blank means "erase"), replacing the one that was already there, or: move the head one tape square to the left (if possible) or right.

Now the *Church-Turing thesis* is the following principle (formulation taken from [14, page 246]):

> *The Turing machine that halts on all inputs is a precise formal notion that corresponds to the naive notion of an "algorithm".*

Finally, the *halting problem* HP is the question whether or not a Turing machine M halts on input string w.

4.2 Enhanced Turing Machine Tape

We consider an enhanced type of Turing Machine Tape (ETMT) over alphabet $\{0, 1, ;\}$. TMT is seen as a reactor, and it is enhanced to ETMT to allow for more powerful programming. A typical state of such a tape is

b	;	;	0	$\hat{1}$	0	1	1	0	;	1	0	1	;	b

where the b stands for *blank*, the semi-colon serves as a separator, and the $\hat{}$ is the head position pointer. The leftmost b represents an indefinite number of blanks to the left [1], and the rightmost b signifies that the tape indefinitely extends to the right. As a consequence, the empty tape (containing only blanks) is represented by

$\hat{\text{b}}$	b

A *bit sequence* on a tape is a sequence of 0 or 1 occurrences of maximal length (so at both ends neighboring either a semicolon or a blank).

We consider the following (service-)instructions:

test:0	*write*:0	*mv*:*left*	*mv*:*begin*
test:1	*write*:1	*mv*:*right*	*dup*
test:*semicolon*	*write*:*semicolon*		
test: b	*write*: b		

with

test:0 (or 1, *semicolon*, b) checks whether the head position points to a 0 (or the other symbol indicated) and returns the appropriate reply (true or false).

write:0 (or 1, *semicolon*) writes the appropriate symbol at head position and returns true.

write: b only works if to the left or the right there is a b already (and returns true), otherwise nothing changes and false is returned.

mv:*left* fails if head is at b and to the left there is a b as well, in this case it returns false and nothing happens; otherwise the head position pointer moves to the left.

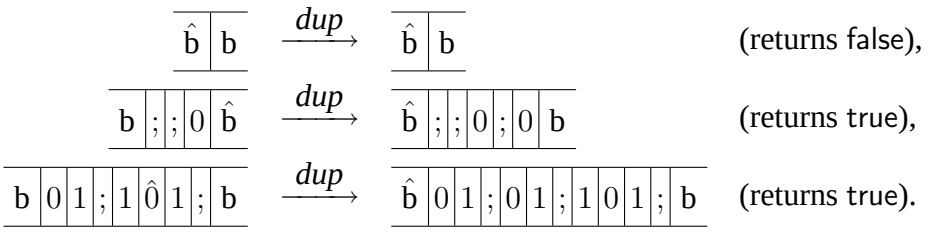
mv:*right* works similar.

mv:*begin* places the head at the left blank and returns true.

dup duplicates the leftmost bit sequence if any exists, and puts the result next to it separated by a semicolon. Furthermore, the head position pointer moves to the

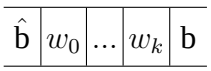
[1] This does not increase the computational power of a Turing Machine (see e.g., [14]).

left blank. Returns true if actual duplication has taken place, and false otherwise. Examples:



The initial configuration of ETMT is

written as ETMT($\hat{\text{b}}$ b), and the configuration that contains sequence $\sigma = w_0...w_k$ with the head at the leftmost blank, i.e.,



is denoted by ETMT($\hat{\text{b}}\,\sigma$ b).

4.3 Programming the Turing Machine

PGLC is the language based on PGA that contains only basic instructions, test instructions and forward and backward jumps. Termination is modeled implicitly in PGLC: a program terminates after the last instruction has been executed and that instruction was no jump into the program, or after a jump outside the program. The sublanguage PGLCi restricts to programs that can safely be concatenated: a forward jump may not exceed the number of subsequent instructions with more than 1 and a backward jump may not exceed the number of preceding instructions, and programs may not end with a test instruction. So $+a;\#1 \in$ PGLCi, but $+a;\#5;b \notin$ PGLCi. Note that behavior extraction on PGLCi is defined by that on PGLC, and that each PGLC-program can be transformed into a behaviorally equivalent PGLCi-program (e.g., $|+a;\#5;b|_{pglc} = |+a;\#2;b|_{pglc}$). So the point of PGLCi is that concatenation of programs yields the expected behavior, e.g., $|+a;\#2;b;c|_{pglc} = c \circ S \trianglelefteq a \trianglerighteq b \circ c \circ S$, while $|+a;\#5;b;c|_{pglc} = S \trianglelefteq a \trianglerighteq b \circ c \circ S$.

We consider the language PGLCi:FMN, where the only focus used will be *etmt* and the basic instructions are those mentioned above for the ETMT. A program in PGLCi:FMN is an ASCII character sequence (see, e.g., [1]), and therefore a sequence of bits. As an example, the character a has 97 as its decimal code, which

is as a byte (sequence of 8 bits) 01100001. The character ";" has 59 as its decimal code, which is as a byte 00111011.

We consider execution of Turing machine programs in an AnArch. For example,

$$\text{pgaEA}(|\textit{etmt.dup};\textit{etmt.mv:right}|_{pglc:fmn},\ \textit{etmt}{:}\text{ETMT}(\,\text{b}\,01;\hat{1}01;\,\text{b}\,))$$

$$\downarrow \tau$$

$$\text{pgaEA}(|\textit{etmt.mv:right}|_{pglc:fmn},\ \textit{etmt}{:}\text{ETMT}(\,\hat{\text{b}}\,01;01;101;\,\text{b}\,))$$

$$\downarrow \tau$$

$$\surd \quad \text{with ETMT's configuration: ETMT}(\,\text{b}\,\hat{0}1;01;101;\,\text{b}\,)$$

where each of the τ-steps ($\xrightarrow{\tau}$) comes from two (abstracted) communications between the current program-fragment $[[|...|_{pglc:fmn}]]$ and the ETMT.

4.4 The Halting Problem

The *Halting Problem* (HP) can in this setting be modeled as follows: a PGLCi:FMN program p halts on the ETMT with initial configuration $\hat{\text{b}}\,w\,\text{b}$ (w a bit sequence), notation $(p, w) \in$ HP, if

$$\text{pgaEA}(|p|_{pglc:fmn},\ \textit{etmt}{:}\text{ETMT}(\,\hat{\text{b}}\,w\,\text{b}\,)) = \tau,$$

as opposed to $\tau^*\delta$ ($= \tau\delta$). After halting (possibly by external means), the tape can be inspected to obtain an output.

Solving the halting problem: we stipulate that program $q \in$ PGLCi:FMN solves the question whether $(p, w) \in$ HP in the following way:

$$\text{pgaEA}(|q|_{pglc:fmn},\ \textit{etmt}{:}\text{ETMT}(\,\hat{\text{b}}\,p;w\,\text{b}\,))$$

where p is stored as a bit sequence always halts, and after halting, the tape configuration is of the form

ETMT($\,\hat{\text{b}}\,0\sigma\,\text{b}\,$) if pgaEA($|p|_{pglc:fmn},\ \textit{etmt}{:}$ETMT($\,\hat{\text{b}}\,w\,\text{b}\,$)) halts, thus $(p, w) \in$ HP,

ETMT($\,\hat{\text{b}}\,1\rho\,\text{b}\,$) if pgaEA($|p|_{pglc:fmn},\ \textit{etmt}{:}$ETMT($\,\hat{\text{b}}\,w\,\text{b}\,$)) halts not, i.e. $(p, w) \notin$ HP,

for some string σ or ρ.

Theorem 5 *The halting problem is unsolvable by means of any program in* PGLCi:FMN.

Proof. Suppose the contrary, i.e., a program $q \in$ PGLCi:FMN exists that solves HP. Consider the following program:

$$s = \textit{etmt.dup}; q; r$$

with $r = \textit{etmt.mv:right}; -\textit{etmt.test}:1; \#0; \textit{etmt.mv:begin}$, and the question $(s, s) \stackrel{?}{\in}$ HP. We show below that both assumptions $(s, s) \in$ HP and $(s, s) \notin$ HP lead to a contradiction. Hence, s cannot exist, and thus q cannot exist.

First, assume that $(s, s) \in$ HP. Then

$$\begin{aligned}
&\text{pgaEA}(|s|_{pglc:fmn},\ \textit{etmt}{:}\text{ETMT}(\,\hat{\text{b}}\; s\; \text{b}\,))\\
&\quad \downarrow \tau \quad (\textit{etmt.dup})\\
&\text{pgaEA}(|q; r|_{pglc:fmn},\ \textit{etmt}{:}\text{ETMT}(\,\hat{\text{b}}\; s; s\; \text{b}\,)).
\end{aligned}$$

Because $q \in$ PGLCi:FMN, the program $q; r$ first executes q (which terminates successfully by assumption) and then starts with the first instruction of r. Thus,

$$\begin{aligned}
&\text{pgaEA}(|q; r|_{pglc:fmn},\ \textit{etmt}{:}\text{ETMT}(\,\hat{\text{b}}\; s; s\; \text{b}\,))\\
&\quad \downarrow \tau \quad (\text{by program } q)\\
&\text{pgaEA}(|r|_{pglc:fmn},\ \textit{etmt}{:}\text{ETMT}(\,\hat{\text{b}}\; 0\sigma\; \text{b}\,))
\end{aligned}$$

for some string σ. The remaining behavior is displayed in Fig. 1, and results in $\text{pgaEA}(|\#0; \textit{etmt.mv:begin}|_{pglc:fmn},\ \textit{etmt}{:}\text{ETMT}(\,\text{b}\,\hat{0}\sigma\,\text{b}\,))$. This last AnArch clearly represents divergence because of the first instruction $\#0$, and therefore $(s, s) \notin$ HP. Contradiction.

Now assume that $(s, s) \notin$ HP. The resulting behavior is displayed in Fig. 2 (for some string ρ). Here the last configuration represents halting, and therefore $(s, s) \in$ HP and again a contradiction occurs.

So our supposition was definitely wrong, i.e., there is no program $q \in$ PGLCi:FMN that solves the halting problem. □

$\mathrm{pgaEA}(|r|_{pglc:fmn},\ \mathit{etmt}{:}\mathrm{ETMT}(\,\hat{\mathrm{b}}\,0\sigma\,\mathrm{b}\,))$

$=$

$\mathrm{pgaEA}(|\mathit{etmt.mv}{:}\mathit{right};-\mathit{etmt.test}{:}1;\#0;\mathit{etmt.mv}{:}\mathit{begin}|_{pglc:fmn},\ \mathit{etmt}{:}\mathrm{ETMT}(\,\hat{\mathrm{b}}\,0\sigma\,\mathrm{b}\,))$

$\downarrow\tau\quad(\mathit{etmt.mv}{:}\mathit{right})$

$\mathrm{pgaEA}(|-\mathit{etmt.test}{:}1;\#0;\mathit{etmt.mv}{:}\mathit{begin}|_{pglc:fmn},\ \mathit{etmt}{:}\mathrm{ETMT}(\,\mathrm{b}\,\hat{0}\sigma\,\mathrm{b}\,))$

$\downarrow\tau\quad(-\mathit{etmt.test}{:}1)$

$\mathrm{pgaEA}(|\#0;\mathit{etmt.mv}{:}\mathit{begin}|_{pglc:fmn},\ \mathit{etmt}{:}\mathrm{ETMT}(\,\mathrm{b}\,\hat{0}\sigma\,\mathrm{b}\,)).$

Fig. 1. Last part of the behavior in the case that $(s,s)\in$ HP in the proof of Thm. 5

$\mathrm{pgaEA}(|s|_{pglc:fmn},\ \mathit{etmt}{:}\mathrm{ETMT}(\,\hat{\mathrm{b}}\,s\,\mathrm{b}\,))$

$\downarrow\tau\quad(\mathit{etmt.dup})$

$\mathrm{pgaEA}(|q;r|_{pglc:fmn},\ \mathit{etmt}{:}\mathrm{ETMT}(\,\hat{\mathrm{b}}\,s;s\,\mathrm{b}\,))$

$\downarrow\tau\quad(\text{by program } q)$

$\mathrm{pgaEA}(|r|_{pglc:fmn},\ \mathit{etmt}{:}\mathrm{ETMT}(\,\hat{\mathrm{b}}\,1\rho\,\mathrm{b}\,))$

$=$

$\mathrm{pgaEA}(|\mathit{etmt.mv}{:}\mathit{right};-\mathit{etmt.test}{:}1;\#0;\mathit{etmt.mv}{:}\mathit{begin}|_{pglc:fmn},\ \mathit{etmt}{:}\mathrm{ETMT}(\,\hat{\mathrm{b}}\,1\rho\,\mathrm{b}\,))$

$\downarrow\tau\quad(\mathit{etmt.mv}{:}\mathit{right})$

$\mathrm{pgaEA}(|-\mathit{etmt.test}{:}1;\#0;\mathit{etmt.mv}{:}\mathit{begin}|_{pglc:fmn},\ \mathit{etmt}{:}\mathrm{ETMT}(\,\mathrm{b}\,\hat{1}\rho\,\mathrm{b}\,))$

$\downarrow\tau\quad(-\mathit{etmt.test}{:}1)$

$\mathrm{pgaEA}(|\mathit{etmt.mv}{:}\mathit{begin}|_{pglc:fmn},\ \mathit{etmt}{:}\mathrm{ETMT}(\,\mathrm{b}\,\hat{1}\rho\,\mathrm{b}\,))$

$\downarrow\tau\quad(\mathit{etmt.mv}{:}\mathit{begin})$

$\surd\quad$ with ETMT's configuration: $\mathit{etmt}{:}\mathrm{ETMT}(\,\hat{\mathrm{b}}\,1\rho\,\mathrm{b}\,).$

Fig. 2. The case that $(s,s)\notin$ HP in the proof of Thm. 5

It is an easy but boring task to program *mv*:*begin* and *dup* in terms of the other instructions, thus obtaining a stronger proof.

Note 3 Namely: *mv*:*begin* is simply programmed by $+\mathit{mv}{:}\mathit{left};\backslash\#1$. Programming the instruction *dup* is slightly more complex, it can for instance be programmed in the following

style:
1. Initialization: a program fragment *Init* that adds two extra semicolons and sets the head pointer to the leftmost bit, followed by
2. Loop: repeatedly copy all bits of the sequence, followed by
3. Exit: remove redundant semicolon and set head pointer (*mv*:*begin*).
We sketch these three program fragments, all of which can be easily programmed with the primitives.

1. Initialize the tape configuration if it contains a bit sequence by adding two semicolons: one following the end of the bit sequence, and one following the leftmost bit, after which the head points at the leftmost bit, e.g.,

$$\mathrm{b}\,;\,;1001\sigma \xrightarrow{\textit{Init}} \mathrm{b}\,;\,;\hat{1};001;\sigma$$

where σ is either a blank, or starts with a semicolon. In case there is no bit sequence, the procedure ends here.

2. Copying can for instance be done with the following two program fragments:
2.1. One, say *write*:$(2,x)$, that writes bit-value x after all bits following the second semicolon from head position, and puts the head just left of the first semicolon following the leftmost bit, e.g.,

$$\mathrm{b}\,;\,;\hat{1};001;\sigma \xrightarrow{\textit{write}:(2,1)} \mathrm{b}\,;\,;\hat{1};001;1\sigma$$

$$\mathrm{b}\,;\,;1\hat{0};01;1\sigma \xrightarrow{\textit{write}:(2,0)} \mathrm{b}\,;\,;1\hat{0};01;10\sigma$$

It is not hard to program *write*:$(2,0)$ and *write*:$(2,1)$ in terms of the other primitive instructions.
2.2. Another program fragment, say *exchange*, that changes the pattern $\hat{x};y$ into $x\hat{y};$ and terminates if y is a semicolon, e.g.,

$$\mathrm{b}\,;\,;\hat{1};001;1\sigma \xrightarrow{\textit{exchange}} \mathrm{b}\,;\,;1\hat{0};01;1\sigma$$

Also this program fragment is easy to program with the primitive instructions.
Now repeatedly perform *write*:$(2,x)$;*exchange* for the appropriate value of x. The loop ends if the leftmost bit sequence is copied, separating the copy with a semicolon, e.g.,

$$\mathrm{b}\,;\,;100\hat{1};\,;1001\sigma \xrightarrow{\textit{exchange}} \mathrm{b}\,;\,;1001\hat{;};1001\sigma$$

3. Exit: remove the semicolon at head position and then execute *mv*:*begin*. This completes the duplication.

As a theorem, the above one (Theorem 5) suffices. From that point of view there is nothing special about the (E)TMT or any of its versions. What we see is that:

(1) For a close relative of the TMT an impossibility result is obtained.
(2) Increasing the instruction set of the ETMT to a 'super' ETMT does not help. The proof goes exactly the same. Computability of these instructions is immaterial. What matters is that the halting problem (HP) is posed about all programs over the instruction set that may be used to program its solution.

(3) The Church-Turing thesis is not used because the result is phrased about PGLCi:FMN programs, and not about 'algorithms' or 'computable methods'. Nevertheless, if it is considered convincing that an effective method can be performed by a certain Turing machine, then it is also obvious that it can be programmed in PGLCi:FMN:
- finite control can be modeled in the program;
- additional instructions only strengthen the expressive power.

This situation changes if we restrict PGLCi:FMN. Note that in our proof we only use the instructions *dup*, *test*:1, *mv*:*right* and *mv*:*begin*. Now if we consider these as the *only* primitive instructions, it is quite clear that the halting problem becomes decidable: call the resulting language PGLCi:FMN$^-$.

Theorem 6 *With the language* PGLCi:FMN$^-$*, the halting problem is decidable.*

Proof. If the tape contains no sequence of bits, each occurrence of *dup* can be replaced by *mv*:*begin* and the tape remains a fixed and finite structure. Execution now either yields a cyclic pattern or stops at $\#0$ or a terminating jump. As there are finitely many combinations of current instruction and head position, halting is decidable.

In the other case, consider some tape configuration that contains a bit sequence and $X \in$ PGLCi:FMN$^-$. Transform X to a canonical form. If this contains no repetition, we are done, otherwise we obtain a program $U;(V)^\omega$ with U and V containing no repetition. Halting on U is decidable: either one of the decisive instructions ! or $\#0$ is to be executed, or execution enters the repeating part V^ω. So, we further consider pgaEA$(|W^\omega|,\ \mathrm{ETMT}(\,\mathrm{b}\,\sigma\hat{x}\rho\,\mathrm{b}\,))$ for some cyclic permutation W of V and some tape configuration. Now, either *dup* occurs at a reachable position in W^ω, or not (can be decided from $|W^\omega|$). In the last case, the tape remains a fixed and finite structure, and iterating W yields a regular behavior, so halting is again decidable. In the other case, the number of *mv*:*right*-instructions in W, say N, limits the number of positions that the head can shift to the right. Consider pgaEA$(|W^N;W^\omega|,\ \mathrm{ETMT}(\,\mathrm{b}\,\sigma\hat{x}\rho\,\mathrm{b}\,))$. Either halting can be decided on W^N, or the repeating part is entered, say X^ω (X again some cyclic permutation of W). We may replace all *dup*-instructions in X^ω by *mv*:*begin* because any further duplication yields an unreachable part at the right end of the tape. So, this case is reduced to the previous one, and halting is again decidable. □

Our objective is to position Turing's impossibility result regarding the assessment of halting properties of program execution as a result about programs rather than machines. The mere requirement that programs of a certain form can decide the halting behavior of all programs of that form leads to a contradiction.

This contradiction can be found in the case of programs for a Turing machine tape (TMT). The argument is significantly simplified if an extended command set for a Turing machine tape is used (ETMT). But then the program notation may be reduced to those features (commands) actually playing a role in the argument and the impossibility result remains but now in a setting where the underlying halting problem is in fact decidable.

We conclude that as a methodological fact about computer programming, the undecidability of the halting problem is an impossibility result which is quite independent of the computational power of the machine models to which it can be applied.

5 Forecasting Reactors and Rational Agents

The halting problem can be investigated without the use of TMT's as a problem regarding the potential capabilities of a reactor serving as a computation forecasting device. In this section we show that restricting to true and false is problematic and introduce a third truth-value. Furthermore, we combine forecasters with 'rational agents', and provide a modeling of the Newcomb paradox. Finally, we model the Prisoner's Dilemma as an analytic architecture.

5.1 Forecasting Reactors

Forecasting is not an obvious concept, the idea that it is to be done by means of a machine even less. We will provide a 'clean' intended interpretation of forecasting and investigate its fate in the context of pgaEA. The use of an AnArch is justified because this story is meant at a conceptual level and is not part of any technical design.

In the previous section it was shown that restricting to true and false is problematic. Therefore we now consider the case that the evaluation of test instructions may yield not only true or false, but also the value M (*meaningless*):

$$|{+}a; X| = \begin{cases} a \circ |\#1; X| & \text{if } a\text{'s execution returns true,} \\ a \circ |\#2; X| & \text{if } a\text{'s execution returns false,} \\ a \circ |\#3; X| & \text{if } a\text{'s execution returns } M, \end{cases}$$

and

$$|{-}a; X| = \begin{cases} a \circ |\#2; X| \text{ if } a\text{'s execution returns } \mathsf{true}, \\ a \circ |\#1; X| \text{ if } a\text{'s execution returns } \mathsf{false}, \\ a \circ |\#3; X| \text{ if } a\text{'s execution returns } M. \end{cases}$$

More information on many-valued logics using true, false and M can be found in [4,7].

We will use *fcu* as the focus pointing to a forecasting unit FCU in the following way: *fcu*.Q will ask the forecaster to reply about its opinion regarding the question Q. At this point the precise phrasing of the requirement on the FCU is essential. In pgaEA($|$*fcu*.$Q; X|$, *fcu*:FCU, $\overline{f_i{:}R_i}$) one expects FCU to reply true if Q is valid as an assertion on pgaEA($|X|$, *fcu*:FCU, $\overline{f_i{:}R_i}$), notation

pgaEA($|X|$, *fcu*:FCU, $\overline{f_i{:}R_i}$) **sat** Q.

More precisely, in

pgaEA($|{+}$*fcu*.$Q; u; X|$, *fcu*:FCU, $\overline{f_i{:}R_i}$)

we expect that

- true is returned if pgaEA($|u; X|$, *fcu*:FCU, $\overline{f_i{:}R_i}$) **sat** Q,
- false is returned if pgaEA($|X|$, *fcu*:FCU, $\overline{f_i{:}R_i}$) ~~**sat**~~ Q,
- M is returned otherwise.

Moreover, in case that both true and false could be returned, preference is given to returning true.

Consider $Q =$ *halting*: when pgaEA($|X|$, *fcu*:FCU, $\overline{f_i{:}R_i}$) **sat** *halting* it will hold along all execution traces (irrespective of any context) that X halts. Then, if a reactor can engage in external communications, the possibility that it will must be taken into account. Moreover, we cannot exclude the possibility that a reactor falls into inaction as a result of such an external communication. Therefore we assume the absence of reactors apart from FCU, and investigate to what extent FCU can be made to provide a useful forecast regarding the *halting*-question.

Theorem 7 *A forecasting reactor* FCU *needs the third truth value* M.

Proof. Consider

pgaEA($|{+}$*fcu*.*halting*$; \#0; !; !|$, *fcu*:FCU).

If true is replied then $\mathrm{pgaEA}(|\#0;!;!|,\ \mathit{fcu}{:}\mathrm{FCU})$ **sat** *halting* which is not true; if false is replied then $\mathrm{pgaEA}(|!;!|,\ \mathit{fcu}{:}\mathrm{FCU})$ **sat** *halting* which is also not true. Thus M should be replied, and

$$\mathrm{pgaEA}(|{+}\mathit{fcu}.\mathit{halting};\#0;!;!|,\ \mathit{fcu}{:}\mathrm{FCU}) \xrightarrow{\tau} \mathrm{pgaEA}(|!|,\ \mathit{fcu}{:}\mathrm{FCU})$$

(which will halt by the way). □

We notice that the FCU may use whatever inspection of other parts of $\mathrm{pgaEA}(...)$. However, it cannot correctly forecast the question with either true or false. Nevertheless, a lot is possible, e.g.:

$$\mathrm{pgaEA}(|{+}\mathit{fcu}.\mathit{halting};!;\#0|,\ \mathit{fcu}{:}\mathrm{FCU})$$

generates reply true,

$$\mathrm{pgaEA}(|{+}\mathit{fcu}.\mathit{halting};\#0;\#0|,\ \mathit{fcu}{:}\mathrm{FCU})$$

generates reply false.

Theorem 8 *A best possible* FCU *for halting can be given for* PGA:FMN *with fcu.halting as its only basic instruction.*

Proof. Let X^{true} be obtained from X by replacing each occurrence of *fcu.halting* by *fcu*.true, the test that always yields true, and let X^{false} be defined similarly (replacement by *fcu*.false). We assume that FCU is deterministic. Consider

$$\mathrm{pgaEA}(|{+}\mathit{fcu}.\mathit{halting};u;X|,\ \mathit{fcu}{:}\mathrm{FCU}).$$

(1) In the case that $\mathrm{pgaEA}(|u^{\mathsf{true}};X^{\mathsf{true}}|,\ \mathit{fcu}{:}\mathrm{FCU})$ **sat** *halting*, we may define that ${+}\mathit{fcu}.\mathit{halting}$ returns true, and as a consequence

$$\mathrm{pgaEA}(|{+}\mathit{fcu}.\mathit{halting};u;X|,\ \mathit{fcu}{:}\mathrm{FCU}) \textbf{ sat } \mathit{halting}.$$

(2) In the case that $\mathrm{pgaEA}(|X^{\mathsf{false}}|,\ \mathit{fcu}{:}\mathrm{FCU})$ **sat** *halting*, we may define that ${+}\mathit{fcu}.\mathit{halting}$ returns false, and as a consequence

$$\mathrm{pgaEA}(|{+}\mathit{fcu}.\mathit{halting};u;X|,\ \mathit{fcu}{:}\mathrm{FCU}) \textbf{ sat } \mathit{halting}.$$

(3) If none of the cases above applies, ${+}\mathit{fcu}.\mathit{halting}$ generates reply M.

E.g., pgaEA$(|+$*fcu.halting*$; \#0; !; !|,$ *fcu*:FCU$)$ will return M though it is going to be halting: by returning M it moves to an instruction from where halting is guaranteed indeed, while replying false would not produce a consistent answer.

It is easy to see that if this definition of replies given by FCU returns M, it cannot be replaced by either true or false. So, FCU is optimal. □

So, a halting forecaster can be built, but it cannot always provide a useful reply. On PGA one can decide whether a useful reply can be given. Given the fact that all practical computing takes place on finite state machines for which PGA is of course sufficient, we conclude this:

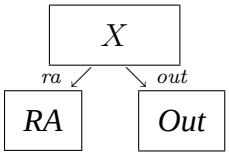
(1) All practical instances of *halting* are decidable, given a pgaEA$(P,\ \overline{f_i{:}R_i})$ with all R_i finite state.
(2) Nevertheless, a halting forecaster cannot work in a flawless fashion, though it can be 'optimal' (i.e., minimizing output M).

If all R_i's are finite state in pgaEA$(P,\ \overline{f_i{:}R_i})$ **sat** *halting* (which is always the case 'in practice'), we find that for a particular AnArch fixing the R_i's the halting problem will be decidable, especially if the AnArch is tailored to fit the constraints of some realistic SynArch.

Of course, one can investigate forecasting *reactors*. Then the question is: what impossibility will one encounter when working in a finite setting? The obvious undecidability result critically depends on one of the reactors being infinite state.

5.2 Reactors Formalizing Rational Agents

We consider a 'rational agent' *RA* with focus ra. The rational agent intends to achieve an objective and acts accordingly. Here is a simple example:

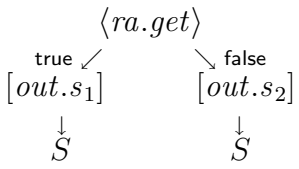


where *Out* has five states $0, 1, 2, 3, 4$ and initially is in state 0. There are four instructions $s_1, ..., s_4$ which all succeed in each state of *Out*, with s_i leaving *Out* in state i for $i \in \{1, 2, 3, 4\}$.

The PGLA:FMN-program X is as follows:

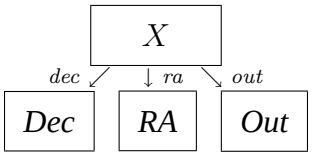
$$+ra.get; \#3; out.s_2; !; out.s_1; !$$

In terms of behavior, $|X|$ can be visualized as follows:



Now the task of *RA* is to make the system terminate with a maximal content of *Out*. *RA* is aware of the contents of program X. In this case, it is clear that the reply false is optimal.

For a second example we add a decision agent *Dec* such that *RA* cannot know which decision *Dec* takes. The focus for *Dec* is dec. The instruction *dec.set* asks *Dec* to take a decision, which it will keep forever, and allows inspection via *dec.get*. An inspection not preceded by a *dec.set* returns M.



The model for *Dec* in concrete process algebra is:

$$[[\textit{Dec}]] = r(set)(t \cdot [[\textit{Dec}^{\mathsf{true}}]] + t \cdot [[\textit{Dec}^{\mathsf{false}}]]),$$

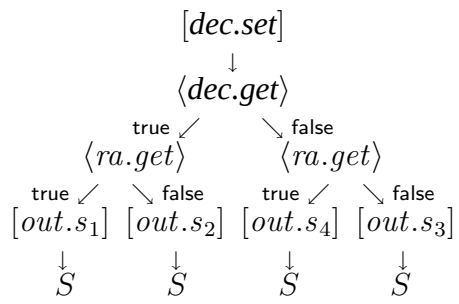
$$[[\textit{Dec}^{\mathsf{true}}]] = (s(\mathsf{true})r(get))^*\delta,$$

$$[[\textit{Dec}^{\mathsf{false}}]] = (s(\mathsf{false})r(get))^*\delta.$$

We consider the following PGLA:FMN-program X' (for readability, some comments are added in the margin):

$$\begin{aligned} X' = \ & \textit{dec.set}; +\textit{dec.get}; \#2; \#7; \\ & +ra.get; \#3; out.s_2; !; out.s_1; !; \ \big| \ \text{(executed if } \textit{dec.get} \text{ returns true)} \\ & +ra.get; \#3; out.s_3; !; out.s_4; ! \ \big| \ \text{(otherwise),} \end{aligned}$$

or in terms of behavior:



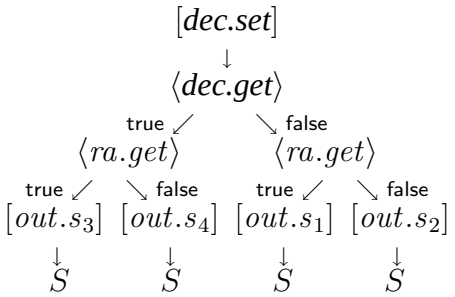
Now both replies true, false are not optimal. If *RA* replies true, this leads to 1 after a positive decision of *Dec* (and false would have given 2), while false is not optimal after a negative decision of *Dec* (giving 3 rather than 4). Therefore it is plausible to return M, but that yields no maximum either (it leaves *Out* in state 0).

5.3 A Newcomb paradox system

In this section we consider the following program, a small modification of the last program in the previous section:

$$
\begin{aligned}
X = \ & \textit{dec.set};\\
& +\textit{dec.get}; \#2; \#7;\\
& \ +ra.get; \#3; out.s_4; !; out.s_3; !;\\
& \ +ra.get; \#3; out.s_2; !; out.s_1; !
\end{aligned}
$$

with behavior



Now, quite independently of *Dec*'s action, it is plausible that *RA* replies false as its best reply. This answer is very robust and covers all possible/conceivable ways for which *RA* might work.

For the next example we introduce the property of pgaEA's that a reactor may be a forecaster of another one:

$$\text{pgaEA}_{forecast:f>g}(\ldots)$$

is as pgaEA$(...)$ but with the additional constraint that reactor f forecasts g. I.e., if $f.get$ returns true (false) then the next $g.get$ will reply true (false) as well.

Consider

$$\text{pgaEA}_{forecast:dec>ra}(|X|,\ dec{:}\textit{Dec}, ra{:}\textit{RA}, out{:}\textit{Out}).$$

Now if *RA* chooses to reply true, *Dec* must have replied true, yielding *Out* in state 3, and if *RA* replies false, *Dec* must have replied false and the yield is 2.

This is a version of Newcomb's paradox, the original form of which has been made famous by Gardner [13, Chapters 13, 14]. [2]

The additional assumption of forecasting reverses the rational answer becoming true (i.e., pick only box B2) instead of false. But the argument for false was completely compelling, making use of case-based reasoning regarding uncertainty about past events.

The Newcomb paradox then arises from the apparently illegal identification of the two following execution environments:

$$\text{pgaEA}(|X|,\ dec{:}\textit{Dec}, ra{:}\textit{RA}, out{:}\textit{Out}) \quad \text{and}$$

$$\text{pgaEA}_{forecast:dec>ra}(|X|\ , dec{:}\textit{Dec}, ra{:}\textit{RA}, out{:}\textit{Out}).$$

In particular, the mistaken view that the second architecture somehow refines the first one by merely providing additional information leads to a contradiction. Thus: adding more information about the component *Dec*, the plausibility of *RA* giving the reply false in order to maximize the contents of *Out* at program termination is lost.

To formalize forecasting between reactors, we use a constant 0 in process algebra (see [2]):

$$\begin{aligned} &x + 0 = x,\\ &x \cdot 0 = 0 \quad \text{provided } x \neq \delta,\\ &0 \cdot x = 0. \end{aligned}$$

[2] A short description based on this source: Given two boxes, B1 which contains $1000 and B2 which contains either nothing or a million dollars, you may pick either B2 or both. However, at some time before the choice is made, an omniscient Being has predicted what your decision will be and filled B2 with a million dollars if he expects you to take it, or with nothing if he expects you to take both.

We write $F_{a>b>c}(X)$ for the following modification of process X: from the first a onwards, replace all b's by 0 until the first occurrence of c. The operation $F_{a>b>c}$ is axiomatized in Table 4.

$F_{a>b>c}(e) = e,$	$F_{a>b>c}(x+y) = F_{a>b>c}(x) + F_{a>b>c}(y),$
$F'_{a>b>c}(e) = 0$ if $e = b,$	$F'_{a>b>c}(x+y) = F'_{a>b>c}(x) + F'_{a>b>c}(y),$
$F'_{a>b>c}(e) = e$ otherwise,	$F_{a>b>c}(e \cdot x) = e \cdot F'_{a>b>c}(x)$ if $e = a,$
$F_{a>b>c}(\delta) = \delta,$	$F_{a>b>c}(e \cdot x) = e \cdot F_{a>b>c}(x)$ otherwise,
$F'_{a>b>c}(\delta) = \delta,$	$F'_{a>b>c}(e \cdot x) = 0$ if $e = b,$
	$F'_{a>b>c}(e \cdot x) = e \cdot x$ if $e = c,$
	$F'_{a>b>c}(e \cdot x) = e \cdot F'_{a>b>c}(x)$ otherwise.

Table 4.
Axioms for $F_{a>b>c}$, where a, b, c, e are actions

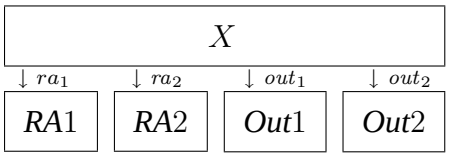
Now forecasting can be formalized as follows:

$$\begin{aligned}
&\text{cpgaEA}_{forecast:dec>ra}(|X|,\ dec{:}\textbf{\textit{Dec}}, ra{:}\textbf{\textit{RA}}, out{:}\textbf{\textit{Out}}) \ = \\
&\quad F_{c_{dec}(\mathsf{true})>c_{ra}(\mathsf{false})>c_{dec}(\mathsf{true})}(\\
&\quad F_{c_{dec}(\mathsf{false})>c_{ra}(\mathsf{true})>c_{dec}(\mathsf{false})}(\text{cpgaEA}(|X|,\ dec{:}\textbf{\textit{Dec}}, ra{:}\textbf{\textit{RA}}, out{:}\textbf{\textit{Out}})))\\
&\text{pgaEA}_{forecast:dec>ra}(|X|,\ dec{:}\textbf{\textit{Dec}}, ra{:}\textbf{\textit{RA}}, out{:}\textbf{\textit{Out}}) \ = \\
&\quad \tau_I(\text{cpgaEA}_{forecast:dec>ra}(|X|,\ dec{:}\textbf{\textit{Dec}}, ra{:}\textbf{\textit{RA}}, out{:}\textbf{\textit{Out}})) \text{ for appropriate } I.
\end{aligned}$$

5.4 Prisoner's Dilemma

A close relative of the above examples is the so-called prisoner's dilemma, which has become very well-known in game theory and economics.

Consider the following situation:

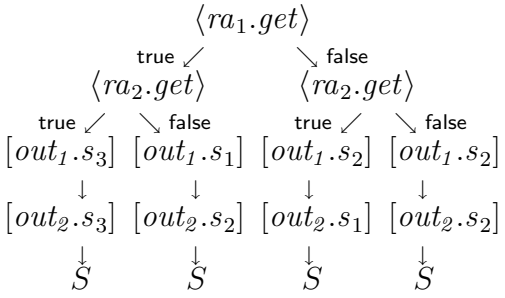


Each rational agent has its own "out" and intends to maximize its own yield, irre-

spective of the other yield. We define the program X by

$$\begin{aligned} X = {} & +ra_1.get; \#2; \#9; \\ & +ra_2.get; \#4; out_1.s_1; out_2.s_2; !; out_1.s_3; out_2.s_3; !; \\ & +ra_2.get; \#4; out_1.s_2; out_2.s_2; !; out_1.s_2; out_2.s_1; ! \end{aligned}$$

or, in a picture, by



Now:

- if *RA*1 and *RA*2 both reply true, both yield the value 3,
- if *RA*1 and *RA*2 both reply false, each gets the value 2,
- if one replies true and the other false, the reply false gets 2 and the reply true yields 1.

As a consequence, in order to exclude the risk of yielding only 1 a sure strategy is to choose false. But in order ot get 3, both *RA*'s must trust one another and choose true, at the same time taking the risk to get only 1. Unable to communicate the *RA*'s may both go for certainty and reply false.

A common application of this is to assume that the reply true complies with the law and false opposes the law. If both comply with the law, both have significant advantage. Complying with the law while others don't is counterproductive, however.

References

[1] ASCII Table and Description. `www.asciitable.com`

[2] J.C.M. Baeten and J.A. Bergstra. Process algebra with a zero object. In: Proceedings CONCUR'90, Amsterdam (J.C.M. Baeten and J.W. Klop, eds.), *Lecture Notes in Computer Science* 458:83–98, Springer-Verlag 1990.

[3] J.A. Bergstra, I. Bethke, and A. Ponse. Process algebra with iteration and nesting. *The Computer Journal*, 37(4):243-258, 1994.

[4] J.A. Bergstra, I. Bethke, and P.H. Rodenburg. A propositional logic with 4 values: true, false, divergent and meaningless. *Journal of Applied Non-Classical Logics*, 5:199-217, 1995.

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

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

[7] J.A. Bergstra and A. Ponse. Bochvar-McCarthy logic and process algebra. *Notre Dame Journal of Formal Logic*, 39(4):464-484, 1998.

[8] J.A. Bergstra and A. Ponse. Register-machine based processes. *Journal of the ACM*, 48(6):1207-1241, 2001.

[9] J.A. Bergstra and A. Ponse. Non-regular iterators in process algebra. *Theoretical Computer Science*, 269 (1-2):203-229, 2001.

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

[11] J.A. Bergstra, A. Ponse, and S.A. Smolka, editors. *Handbook of Process Algebra*, Elsevier Science, 2001.

[12] F. Cohen. Computer viruses - theory and experiments, 1984. `http://vx.netlux.org/lib/afc01.html`.

[13] M. Gardner. *Knotted Doughnuts and Other Mathematical Entertainments*. New York: W. H. Freeman, 1986.

[14] H.R. Lewis and C.H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, 1981.

[15] H. Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill Book Co., 1967.

[16] A. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. Lond. Math. Soc. Ser. 2*, 42:230–265, 1937. Corrections: *ibid* 43:544–546, 1937.