# On the contribution of backward jumps to instruction sequence expressiveness

Bergstra, J.A.; Bethke, I.

**Publication date**
2010
**Document Version**
Submitted manuscript

# On the contribution of backward jumps to instruction sequence expressiveness

Jan A. Bergstra
Inge Bethke

Section Theory of Computer Science, Informatics Institute, University of Amsterdam
URL: www.science.uva.nl/~{janb,inge}

June 1, 2010

**Abstract**

We investigate the expressiveness of backward jumps in a framework of formalized sequential programming called *program algebra*. We show that—if expressiveness is measured in terms of the computability of partial Boolean functions—then backward jumps are superfluous. If we, however, want to prevent explosion of the length of programs, then backward jumps are essential.

## 1 Introduction

We take the view that sequential programs are in essence instruction sequences which leads to an algebraic approach to the formal description of the semantics of programming languages also known as *program algebra*. It is a framework that permits algebraic reasoning about programs and has been investigated in various settings (see e.g. [3, 9, 10, 11, 16]). Here the notion of program algebra refers to the concept introduced in [3] where the behaviour of a program is taken for a *thread*, i.e. a form of process that is tailored to the description of the behaviour of a deterministic sequential program under execution.

In addition to basic, test and termination instructions, program algebra considers two sorts of unconditional jump instructions: *forward* and *backward* jumps. If only forward jumps are permitted, then threads that perform an infinite sequence of actions are excluded. In other words, programs for which the execution goes on indefinitely cannot be expressed. However, in a setting with backward jump instructions also every *regular* infinite thread—i.e. every infinite, finite state process—can be described by a finite sequence of primitive instructions.

The aim of this paper is to give an indication of the expressiveness of backward jumps, where expressiveness is measured in terms of the Boolean partial functions that can be computed with the aid of instruction sequences. As it will turn out every partial Boolean

function can be computed without backward jumps. Thus, semantically we can do without backward jumps. However, if we want to avoid an explosion of the length on instruction sequences, then backward jumps are essential.

This paper is organized as follows. Section 2 briefly recalls the program notation $\mathrm{PGLB}_{bt}$ and its accompanying thread algebra. In Section 3 we review *services* and the interactions of services with threads. Section 4 investigates the expressiveness of backward jumps.

## 2 Instruction sequences and regular threads

In this section, we briefly recall the program notation $\mathrm{PGLB}_{\mathrm{bt}}$ and its accompanying thread algebra. PGLB is a notation for instruction sequences and belongs to a hierarchy of program notations in the program algebra PGA introduced in [3] (see also [15]). $\mathrm{PGLB}_{\mathrm{bt}}$ is PGLB with the termination instruction ! refined into two Boolean termination instructions $!\mathtt{t}, !\mathtt{f}$ (see also [5, 6, 7]). Both PGLB and $\mathrm{PGLB}_{\mathrm{bt}}$ are close to existing assembly languages and have relative jump instructions.

Assume $A$ is a set of constants with typical elements $\mathtt{a}, \mathtt{b}, \mathtt{c}, \ldots$. $\mathrm{PGLB}_{\mathrm{bt}}(A)$ instruction sequences are then of the following form ($\mathtt{a} \in A$, $l \in \mathbb{N}$):

$$I ::= \mathtt{a} \mid +\mathtt{a} \mid -\mathtt{a} \mid \#l \mid \backslash\#l \mid !\mathtt{t} \mid !\mathtt{f} \mid I; I.$$

The first seven forms above are called *primitive instructions*. These are

1. *basic instructions* $\mathtt{a}$ which prescribe actions that are considered indivisible and executable in finite time, and which return upon execution a Boolean reply value,

2.-3. *test instructions* obtained from basic instructions by prefixing them with either $+$ (positive test instruction) or $-$ (negative test instruction) which control subsequent execution via the reply of their execution,

4.-5. *jump instructions* $\#l, \backslash\#l$ which prescribe to jump $l$ instructions forward and backward, respectively—if possible; otherwise deadlock occurs—and generate no observable behavior, and

6.-7. the *termination instructions* $!\mathtt{t}, !\mathtt{f}$ which prescribe successful termination and in doing so deliver the Boolean value $\mathtt{t}$ and $\mathtt{f}$, respectively.

Complex instruction sequences are obtained from primitive instructions using *concatenation*: if $I$ and $J$ are instruction sequences, then so is

$$I; J$$

which is the instruction sequence that lists $J$'s primitive instructions right after those of $I$. We denote by $\mathcal{IS}(A)$ the set of $\mathrm{PGLB}_{\mathrm{bt}}(A)$ instruction sequences.

Thread algebra is the behavioural semantics for PGA and was introduced in e.g. [1, 3] under the name Polarized Process Algebra.

In the setting of $\mathrm{PGLB}_{\mathrm{bt}}(A)$, finite threads are defined inductively by:

$$\begin{array}{rcl} \mathsf{S}+ & - & \text{the termination thread with positive reply,} \\ \mathsf{S}- & - & \text{the termination thread with negative reply,} \\ \mathsf{D} & - & \textit{inaction} \text{ or } \textit{deadlock}\text{, the inactive thread,} \\ T \trianglelefteq \mathtt{a} \trianglerighteq T' & - & \text{the } \textit{postconditional composition} \text{ of } T \text{ and } T' \text{ for action } \mathtt{a}, \\ & & \text{where } T \text{ and } T' \text{ are finite threads and } \mathtt{a} \in A. \end{array}$$

The behaviour of the thread $T \trianglelefteq \mathtt{a} \trianglerighteq T'$ starts with the *action* $\mathtt{a}$ and continues as $T$ upon reply $\mathtt{t}$ to $\mathtt{a}$, and as $T'$ upon reply $\mathtt{f}$. Note that finite threads always end in $\mathsf{S}+, \mathsf{S}-$ or $\mathsf{D}$. We use *action prefix* $\mathtt{a} \circ T$ as an abbreviation for $T \trianglelefteq \mathtt{a} \trianglerighteq T$ and take $\circ$ to bind strongest.

Infinite threads are obtained by guarded recursion. A *guarded recursive specification* is a set of recursion equations $E = \{E_i = T_i \mid i \in I\}$ where each $T_i$ is of the form $\mathsf{S}+, \mathsf{S}-, \mathsf{D}$ or $T \trianglelefteq \mathtt{a} \trianglerighteq T'$ with $T, T'$ process terms with variables from $\{E_i \mid i \in I\}$. A *regular* thread is a finite state thread in which infinite paths may occur. Regular threads correspond to finite guarded recursive specifications, i.e. guarded recursive specifications with a finite number of recursive equations. To reason about infinite threads, we assume the *Approximation Induction Principle*

$$\bigwedge_{n \geq 0} \pi_n(T) = \pi_n(T') \Rightarrow T = T' \qquad (AIP).$$

*AIP* identifies two threads if their approximations up to any finite depth are identical. The approximation up to depth $n$ of a thread is obtained by cutting it off after $n$ performed actions. In *AIP*, the approximation up to depth $n$ is phrased in terms of the *projection* operator $\pi_n$ which is defined by

1. $\pi_0(T) = \mathsf{D}$,
2. $\pi_{n+1}(\mathsf{S}+) = \mathsf{S}+$,
3. $\pi_{n+1}(\mathsf{S}-) = \mathsf{S}-$,
4. $\pi_{n+1}(\mathsf{D}) = \mathsf{D}$, and
5. $\pi_{n+1}(T \trianglelefteq \mathtt{a} \trianglerighteq T') = \pi_n(T) \trianglelefteq \mathtt{a} \trianglerighteq \pi_n(T')$

for $n \in \mathbb{N}$. Every infinite thread $T$ can be identified with its *projective sequence* $(\pi_n(T))_{n \in \mathbb{N}}$.

Upon its execution, a basic or test instruction yields the equally named action in a post conditional composition. Thread extraction on $\mathrm{PGLB}_{\mathrm{bt}}(A)$, notation $|X|$ with $X \in \mathcal{IS}(A)$, is defined by

$$|X| = |1, X|$$

where $|\ ,\ |$ in turn is defined by the equations given in Table 1. In particular, note that upon the execution of a positive test instruction $+\mathtt{a}$, the reply $\mathtt{t}$ to $\mathtt{a}$ prescribes to continue with the next instruction and $\mathtt{f}$ to skip the next instruction and to continue with the instruction thereafter; if no such instruction is available, deadlock occurs. For the execution of a negative test instruction $-\mathtt{a}$, subsequent execution is prescribed by the complementary replies.

$$
\begin{array}{lcll}
|i,u_1;\ldots;u_k| & = & \mathsf{D} & \text{if } i=0 \text{ or } k<i \\
|i,u_1;\ldots;u_k| & = & \mathtt{a}\circ|i+1,u_1;\ldots;u_k| & \text{if } u_i=\mathtt{a} \\
|i,u_1;\ldots;u_k| & = & |i+1,u_1;\ldots;u_k| \trianglelefteq \mathtt{a} \trianglerighteq |i+2,u_1;\ldots;u_k| & \text{if } u_i=+\mathtt{a} \\
|i,u_1;\ldots;u_k| & = & |i+2,u_1;\ldots;u_k| \trianglelefteq \mathtt{a} \trianglerighteq |i+1,u_1;\ldots;u_k| & \text{if } u_i=-\mathtt{a} \\
|i,u_1;\ldots;u_k| & = & |i+l,u_1;\ldots;u_k| & \text{if } u_i=\#l \\
|i,u_1;\ldots;u_k| & = & |i-l,u_1;\ldots;u_k| & \text{if } u_i=\backslash\#l \text{ and } i>l \\
|i,u_1;\ldots;u_k| & = & |0,u_1;\ldots;u_k| & \text{if } u_i=\backslash\#l \text{ and } i\leq l \\
|i,u_1;\ldots;u_k| & = & \mathsf{S}+ & \text{if } u_i=!\mathtt{t} \\
|i,u_1;\ldots;u_k| & = & \mathsf{S}- & \text{if } u_i=!\mathtt{f}
\end{array}
$$

Table 1: Equations for thread extraction, where $\mathtt{a}$ ranges over the basic instructions and $i,k,l\in\mathbb{N}$

If we add the rule

$$|i,,u_1;\ldots;u_k| = \mathsf{D} \text{ if } u_i \text{ is the beginning of an infinite jump chain}$$

then thread extraction on $\mathrm{PGLB}_{bt}(A)$ yields regular threads. Conversely, every regular thread corresponds to a $\mathrm{PGLB}_{bt}(A)$ instruction sequence after thread extraction.

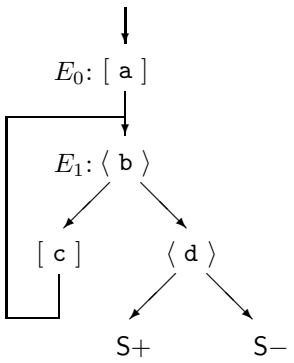
**Example 2.1.** We consider the $\mathrm{PGLB}_{bt}(A)$ instruction sequence

$$X = \mathtt{a};+\mathtt{b};\#2;\#3;\mathtt{c};\backslash\#4;+\mathtt{d};!\mathtt{t};!\mathtt{f}.$$

Thread extraction of $X$ yields the regular thread

$$\begin{aligned} E_0 &= \mathtt{a}\circ E_1 \\ E_1 &= \mathtt{c}\circ E_1 \trianglelefteq \mathtt{b} \trianglerighteq (\mathsf{S}+ \trianglelefteq \mathtt{d} \trianglerighteq \mathsf{S}-) \end{aligned}$$

A picture of this thread is




Here $[\mathtt{a}]$ corresponds to action prefix and $\langle\mathtt{a}\rangle$ to postconditional composition with a left hand

vector continuing the path in case of a positive reply and a right hand vector in case of a negative reply.

This thread can also given by the projective sequence $(\pi_n(E_0))_{n\in\mathbb{N}}$ where

$$\begin{array}{lcl}\pi_0(E_0) & = & \mathsf{D}\\ \pi_1(E_0) & = & \mathtt{a}\circ\mathsf{D}\\ \pi_2(E_0) & = & \mathtt{a}\circ\mathtt{b}\circ\mathsf{D}\\ \pi_3(E_0) & = & \mathtt{a}\circ(\mathtt{c}\circ\mathsf{D}\trianglelefteq\mathtt{b}\trianglerighteq\mathtt{d}\circ\mathsf{D})\end{array}$$

and $\pi_{n+4}(E_0) = \mathtt{a}\circ(\mathtt{c}\circ\pi_{n+1}(E_1)\trianglelefteq\mathtt{b}\trianglerighteq(\mathsf{S}+\trianglelefteq\mathtt{d}\trianglerighteq\mathsf{S}-))$ where

$$\begin{array}{lcl}\pi_0(E_1) & = & \mathsf{D}\\ \pi_1(E_1) & = & \mathtt{b}\circ\mathsf{D}\end{array}$$

and $\pi_{n+2}(E_1) = \mathtt{c}\circ\pi_n(E_1)\trianglelefteq\mathtt{b}\trianglerighteq(\mathsf{S}+\trianglelefteq\mathtt{d}\trianglerighteq\mathsf{S}-)$.

For basic information on thread algebra we refer to [2, 15]; more advanced matters, such as an operational semantics for thread algebra, are discussed in [4].

# 3 Services

Services process certain methods which may involve a change of state, and produce reply values. In the sequel, we let $\mathcal{M}$ be an arbitrary but fixed set of *methods* and $\mathcal{R} = \{\mathtt{t},\mathtt{f},\mathtt{d}\}$ be the set of *reply values* with $\mathtt{d}$ the divergent value which is neither true nor false.

A *service* $\mathbb{S}$ consists of

1. a set $S$ of *states* in which the service may be,
2. an *effect* function $\mathit{eff} : \mathcal{M}\times S\to S$ that gives for each method $m$ and state $s$ the resulting state after processing $m$,
3. a *yield* function $\mathit{yld} : \mathcal{M}\times S\to\mathcal{R}$ that gives for each method $m$ and state $s$ the resulting reply after processing $m$, and
4. an *initial state* $s_0\in S$

satisfying the condition

(†) $\quad\exists s\in S\,\forall m\in\mathcal{M}\,(\ \mathit{yld}(m,s)=\mathtt{d}\ \&\ \forall s'\in S\,(\ \mathit{yld}(m,s')=\mathtt{d}\Rightarrow\mathit{eff}(m,s')=s\ )\ )$.

Given a service $\mathbb{S} = \langle S,\mathit{eff},\mathit{yld},s_0\rangle$ and a method $m\in\mathcal{M}$,

5. the *derived service of* $\mathbb{S}$ *after processing* $m$, $\frac{\partial}{\partial m}\mathbb{S}$, is defined by

$$\frac{\partial}{\partial m}\mathbb{S} = \langle S,\mathit{eff},\mathit{yld},\mathit{eff}(m,s_0)\rangle$$

6. the *reply of* $\mathbb{S}$ *after processing* $m$, $\mathbb{S}(m)$, is defined by $\mathbb{S}(m) = yld(m, s_0)$.

When a request is made to service $\mathbb{S}$ to process method $m$ then

7. if $\mathbb{S}(m) \neq \mathtt{d}$, then the service processes $m$, produces the reply $\mathbb{S}(m)$, and proceeds as $\frac{\partial}{\partial m}\mathbb{S}$, but
8. if $\mathbb{S}(m) = \mathtt{d}$, then the service rejects the request and proceeds as a service that rejects any request to process a method.

An *empty* service $\mathbb{S}$ is a service that is unable to process any method, i.e. $\mathbb{S}(m) = \mathtt{d}$ for all $m \in \mathcal{M}$. Given (†), we can identify all empty services and denote it $\delta$. A set of services is called *closed* if it contains the empty service and is closed under $\frac{\partial}{\partial m}$ for all $m \in \mathcal{M}$.

**Example 3.1.** Given the set of methods $\mathcal{M} = \{\mathtt{set{:}t}, \mathtt{set{:}f}, \mathtt{get}\}$, we consider the set of services $\mathcal{B} = \{B(x) \mid x \in \mathcal{R}\}$ of Boolean registers with initial values $\mathtt{t}$, $\mathtt{f}$ and $\mathtt{d}$, respectively. Here for $x \in \mathcal{R}$, $B(x) = \langle \mathcal{R}, \mathit{eff}, yld, x\rangle$ where

$$\mathit{eff}(\mathtt{set{:}t}, x) = \begin{cases} \mathtt{t} & \text{if } x = \mathtt{f}, \text{ and} \\ x & \text{otherwise} \end{cases}$$

$$\mathit{eff}(\mathtt{set{:}f}, x) = \begin{cases} \mathtt{f} & \text{if } x = \mathtt{t}, \text{ and} \\ x & \text{otherwise,} \end{cases}$$

and $\mathit{eff}(\mathtt{get}, x) = x$; for $m \in \mathcal{M}$, $yld(m, x) = \mathtt{t}$ if $x \in \{\mathtt{t}, \mathtt{f}\}$ and $yld(m, \mathtt{d}) = \mathtt{d}$. Observe that $\mathcal{B}$ is closed with $\delta = B(\mathtt{d})$ and

$$\tfrac{\partial}{\partial \mathtt{set{:}t}}B(\mathtt{t}) = B(\mathtt{t}), \quad \tfrac{\partial}{\partial \mathtt{set{:}t}}B(\mathtt{f}) = B(\mathtt{t}), \quad \tfrac{\partial}{\partial \mathtt{set{:}t}}B(\mathtt{d}) = B(\mathtt{d}),$$

$$\tfrac{\partial}{\partial \mathtt{set{:}f}}B(\mathtt{t}) = B(\mathtt{f}), \quad \tfrac{\partial}{\partial \mathtt{set{:}f}}B(\mathtt{f}) = B(\mathtt{f}), \quad \tfrac{\partial}{\partial \mathtt{set{:}f}}B(\mathtt{d}) = B(\mathtt{d}),$$

and $\frac{\partial}{\partial \mathtt{get}}B(x) = B(x)$ for $x \in \mathcal{R}$.

A *service family* is a set of services uniquely named by a fixed but arbitray set $\mathcal{F}$ of *foci*. $\emptyset$ denotes the empty service family, and for $f \in \mathcal{F}$ and service $\mathbb{S}$, $f.\mathbb{S}$ denotes the singleton service family consisting of the named service $f.\mathbb{S}$. $\oplus$ denotes the binary composition operator which forms the union of service families under the provision that named services with the same name collapse to the empty service with that name. For $F \subseteq \mathcal{F}$, $\partial_F$ denotes the unary encapsulation operator which removes the named services with a name in $F$ from a given service family. The axioms for service families are given in Table 2.

Let $A = \{f.m \mid f \in \mathcal{F},\ m \in \mathcal{M}\} \cup \{\mathtt{tau}\}$ where $\mathtt{tau}$ denotes a basic internal action which does not have any side effects and always replies true. Then a thread may make *use* of services by performing a basic action for the purpose of requesting a named service to process a method and to return a *reply* value at completion of the processing of the method. In the sequel, we combine threads with services and extend the combination with the two operators / and ! which relate to this kind of interaction between threads and services.

| | | | | |
|---|---|---|---|---|
| $u \oplus \emptyset = u$ | SFC1 | $\partial_F(\emptyset) = \emptyset$ | | SFE1 |
| $u \oplus v = v \oplus u$ | SFC2 | $\partial_F(f.\mathbb{S}) = \emptyset$ | if $f \in F$ | SFE2 |
| $(u \oplus v) \oplus w = u \oplus (v \oplus w)$ | SFC3 | $\partial_F(f.\mathbb{S}) = f.\mathbb{S}$ | if $f \notin F$ | SFE3 |
| $f.\mathbb{S} \oplus f.\mathbb{S}' = f.\delta$ | SFC4 | $\partial_F(u \oplus v) = \partial_F(u) \oplus \partial_F(v)$ | | SFE4 |

Table 2: Axioms for binary composition and encapsulation of service families with $f \in \mathcal{F}$, $F \subseteq \mathcal{F}$ and services $\mathbb{S}, \mathbb{S}'$.

The thread denoted by a closed term of the form $T/\mathcal{S}$ is the thread that results from processing the method of each basic action with a focus of the service family denoted by $\mathcal{S}$ that the thread denoted by $T$ performs, where the processing is done by the service in that service family with the focus of the basic action as its name. When the method of a basic action performed by a thread is processed by a service, the service changes in accordance with the method concerned, and affects the thread as follows: the basic action turns into the internal action $\mathtt{tau}$ and the two ways to proceed reduce to one on the basis of the reply value produced by the service. The value denoted by a closed term of the form $T!\mathcal{S}$ is the Boolean value that the thread denoted by $T/\mathcal{S}$ delivers at its termination, and the value $\mathtt{d}$ if it does not terminate. The axioms for the use and the reply operator were first given in [6] and are listed in Tables 3, and 4. In their original version, the axiomatizations contain the axioms U3 and R3 concerning the use and reply of the unpolarized termination $\mathsf{S}$ with service families. Since we only consider boolean termination, we have omitted these axioms.

| | | |
|---|---|---|
| $\mathsf{S}+/u = \mathsf{S}+$ | | U1 |
| $\mathsf{S}-/u = \mathsf{S}-$ | | U2 |
| $\mathsf{D}/u = \mathsf{D}$ | | U4 |
| $(\mathtt{tau} \circ x)/u = \mathtt{tau} \circ (x/u)$ | | U5 |
| $(x \trianglelefteq f.m \trianglerighteq y)/\partial_{\{f\}}(u) = (x/\partial_{\{f\}}(u)) \trianglelefteq f.m \trianglerighteq (y/\partial_{\{f\}}(u))$ | | U6 |
| $(x \trianglelefteq f.m \trianglerighteq y)/(f.\mathbb{S} \oplus \partial_{\{f\}}(u)) = \mathtt{tau} \circ (x/(f.\frac{\partial}{\partial m}\mathbb{S} \oplus \partial_{\{f\}}(u)))$ | if $\mathbb{S}(m) = \mathtt{t}$ | U7 |
| $(x \trianglelefteq f.m \trianglerighteq y)/(f.\mathbb{S} \oplus \partial_{\{f\}}(u)) = \mathtt{tau} \circ (y/(f.\frac{\partial}{\partial m}\mathbb{S} \oplus \partial_{\{f\}}(u)))$ | if $\mathbb{S}(m) = \mathtt{f}$ | U8 |
| $(x \trianglelefteq f.m \trianglerighteq y)/(f.\mathbb{S} \oplus \partial_{\{f\}}(u)) = \mathsf{D}$ | if $\mathbb{S}(m) = \mathtt{d}$ | U9 |

Table 3: Axioms for the use operator with $f \in \mathcal{F}$, $m \in \mathcal{M}$ and service $\mathbb{S}$

**Example 3.2.** We continue with Example 3.1 and put $\mathcal{F} = \mathbb{N}$. We let $Eq(1,2)$ be the

| | | |
|---|---|---|
| $\mathsf{S}{+}!u = \mathtt{t}$ | | R1 |
| $\mathsf{S}{-}!u = \mathtt{f}$ | | R2 |
| $\mathsf{D}!u = \mathtt{d}$ | | R4 |
| $(\mathtt{tau} \circ x)!u = x!u$ | | R5 |
| $(x \trianglelefteq f.m \trianglerighteq y)!\partial_{\{f\}}(u) = \mathtt{d}$ | | R6 |
| $(x \trianglelefteq f.m \trianglerighteq y)!(f.\mathbb{S} \oplus \partial_{\{f\}}(u)) = x!(f.\frac{\partial}{\partial m}\mathbb{S} \oplus \partial_{\{f\}}(u))$ | if $\mathbb{S}(m) = \mathtt{t}$ | R7 |
| $(x \trianglelefteq f.m \trianglerighteq y)!(f.\mathbb{S} \oplus \partial_{\{f\}}(u)) = y!(f.\frac{\partial}{\partial m}\mathbb{S} \oplus \partial_{\{f\}}(u))$ | if $\mathbb{S}(m) = \mathtt{f}$ | R8 |
| $(x \trianglelefteq f.m \trianglerighteq y)!(f.\mathbb{S} \oplus \partial_{\{f\}}(u)) = \mathtt{d}$ | if $\mathbb{S}(m) = \mathtt{d}$ | R9 |

Table 4: Axioms for the reply operator with $f \in \mathcal{F}$, $m \in \mathcal{M}$ and service $\mathbb{S}$

$\mathrm{PGLB}_{bt}(A)$ instruction sequence

$$+1.\mathtt{get}; \#2; \#4; +2.\mathtt{get}; !\mathtt{t}; !\mathtt{f}; -2.\mathtt{get}; \backslash\#3; \backslash\#3$$

which intuitively describes a finite thread that compares 2 Boolean registers and returns the reply $\mathtt{t}$ if their values are not divergent and equal, $\mathtt{f}$ if their values are not divergent but different, and $\mathtt{d}$ otherwise. Indeed, formalizing this interaction in the setting of services we put $\mathcal{S} = 1.B(b_1) \oplus 2.B(b_2)$ and compute

$$\begin{aligned} |Eq(1,2)|!\mathcal{S} &= ((\mathsf{S}{+} \trianglelefteq 2.\mathtt{get} \trianglerighteq \mathsf{S}{-}) \trianglelefteq 1.\mathtt{get} \trianglerighteq (\mathsf{S}{-} \trianglelefteq 2.\mathtt{get} \trianglerighteq \mathsf{S}{+}))!\mathcal{S} \\ &= \begin{cases} (\mathsf{S}{+} \trianglelefteq 2.\mathtt{get} \trianglerighteq \mathsf{S}{-})!\mathcal{S} & \text{if } b_1 = \mathtt{t}, \\ (\mathsf{S}{-} \trianglelefteq 2.\mathtt{get} \trianglerighteq \mathsf{S}{+})!\mathcal{S} & \text{if } b_1 = \mathtt{f}, \text{ and} \\ \mathtt{d} & \text{if } b_1 = \mathtt{d}. \end{cases} \\ &= \begin{cases} \mathtt{t} & \text{if } b_1 = b_2 \neq \mathtt{d}, \\ \mathtt{f} & \text{if } \mathtt{d} \neq b_1 \neq b_2 \neq \mathtt{d}, \text{ and} \\ \mathtt{d} & \text{if } b_1 = \mathtt{d} \text{ or } b_2 = \mathtt{d}. \end{cases} \end{aligned}$$

We let $E(m,n)$ be the generic equality test for the registers $b_m, b_n$ and

$$E(1,2,3) = +1.\mathtt{get}; \#2; \#4; -2.\mathtt{get}; !\mathtt{f}; \#4; +2.\mathtt{get}; \backslash\#3; 0.\mathtt{set}{:}\mathtt{f}; E(0,3)$$

the lazy equality test of 3 registers which stores an intermediate result in the auxiliary

register $b_0$. Observe that $|Eq(1,2,3)|/0.B(\mathtt{t})$

$$= \quad ((|E(0,3)| \trianglelefteq 2.\mathtt{get} \trianglerighteq \mathsf{S}-) \trianglelefteq 1.\mathtt{get} \trianglerighteq (\mathsf{S}- \trianglelefteq 2.\mathtt{get} \trianglerighteq (0.\mathtt{set{:}f} \circ |E(0,3)|)))/0.B(\mathtt{t})$$

$$= \begin{cases} (|E(0,3)| \trianglelefteq 2.\mathtt{get} \trianglerighteq \mathsf{S}-)/0.B(\mathtt{t}) & \text{if } b_1 = \mathtt{t}, \\ (\mathsf{S}- \trianglelefteq 2.\mathtt{get} \trianglerighteq (0.\mathtt{set{:}f} \circ |E(0,3)|))/0.B(\mathtt{t}) & \text{if } b_1 = \mathtt{f}, \\ \mathsf{D} & \text{if } b_1 = \mathtt{d}, \end{cases}$$

$$= \begin{cases} |E(0,3)|/0.B(\mathtt{t}) & \text{if } b_1 = \mathtt{t} = b_2, \\ (0.\mathtt{set{:}f} \circ |E(0,3))|/0.B(\mathtt{t}) & \text{if } b_1 = \mathtt{f} = b_2, \\ \mathsf{S}- & \text{if } \mathtt{d} \neq b_1 \neq b_2 \neq \mathtt{d}, \\ \mathsf{D} & \text{if } b_1 = \mathtt{d} \text{ or } b_2 = \mathtt{d}, \end{cases}$$

$$= \begin{cases} |E(0,3)|/0.B(\mathtt{t}) & \text{if } b_1 = \mathtt{t} = b_2, \\ \mathtt{tau} \circ (|E(0,3)|/0.B(\mathtt{f})) & \text{if } b_1 = \mathtt{f} = b_2, \\ \mathsf{S}- & \text{if } \mathtt{d} \neq b_1 \neq b_2 \neq \mathtt{d}, \\ \mathsf{D} & \text{if } b_1 = \mathtt{d} \text{ or } b_2 = \mathtt{d}, \end{cases}$$

$$= \begin{cases} \mathtt{tau} \circ \mathsf{S}+ & \text{if } b_1 = \mathtt{t} = b_2 = b_3, \\ \mathtt{tau} \circ \mathsf{S}- & \text{if } b_1 = \mathtt{t} = b_2 \text{ and } b_3 = \mathtt{f}, \\ \mathtt{tau} \circ \mathtt{tau} \circ \mathsf{S}+ & \text{if } b_1 = \mathtt{f} = b_2 = b_3, \\ \mathtt{tau} \circ \mathtt{tau} \circ \mathsf{S}- & \text{if } b_1 = \mathtt{f} = b_2 \text{ and } b_3 = \mathtt{t}, \\ \mathsf{S}- & \text{if } \mathtt{d} \neq b_1 \neq b_2 \neq \mathtt{d}, \text{ and} \\ \mathsf{D} & \text{if } b_1 = \mathtt{d} \text{ or } b_2 = \mathtt{d} \text{ or } b_1 = b_2 \neq \mathtt{d} = b_3. \end{cases}$$

Thus

$$(|E(1,2,3)|/0.B(\mathtt{t}))! \oplus_{i=1}^{3} B(b_i) \quad = \quad \begin{cases} \mathtt{t} & \text{if } b_1 = b_2 = b_3 \neq \mathtt{d}, \\ \mathtt{d} & \text{if } b_1 = \mathtt{d} \text{ or } b_2 = \mathtt{d} \text{ or } b_1 = b_2 \neq \mathtt{d} = b_3, \\ \mathtt{f} & \text{otherwise.} \end{cases}$$

Here $\oplus_{i=1}^{3} B(b_i)$ abbreviates the service family $1.B(b_1) \oplus 2.B(b_2) \oplus 3.B(b_3)$. An equality test for 3 registers can be written in several ways: e.g. without backward jumps by replacing the jump $\backslash\#3$ by $!\mathtt{f}$. Also the use of an auxiliary register can be omitted. We shall come back to this issue in Proposition 4.2.

In the case of regular threads, one can show that projection distributes over use, i.e. that $\pi_n(T/\mathcal{S}) = \pi_n(T)/\mathcal{S}$ for $n \in \mathbb{N}$. It then follows from the Approximation Induction Principle that

$$\bigwedge_{n \geq 0} \pi_n(T)/\mathcal{S} = \pi_n(T')/\mathcal{S}' \Rightarrow T/\mathcal{S} = T'/\mathcal{S}'.$$

For more information about services we refer to [6, 8].

# 4 Backward jumps

Backward jumps $\backslash\#l$ ($l \in \mathbb{N}$) are of obvious importance for constructing instruction sequences with loops. Now one may ask how vital are backward jumps? Consider $\mathtt{a};\backslash\#1$—a $\mathrm{PGLB}_{bt}(A)$ instruction sequence which prescribes the execution of the atomic action $\mathtt{a}$ followed by a backward jump of length 1. This instruction sequence produces the thread $T$ with $T = \mathtt{a} \circ T$—a thread that performs the action $\mathtt{a}$ followed by a recursive invocation of the thread. Clearly no $X \in \mathcal{IS}(A)$ can produce a thread with an unbounded number of successive $\mathtt{a}$'s. Thus backward jumps add to the expressiveness of $\mathrm{PGLB}_{bt}(A)$.

In the field of expressiveness and computational complexity one classifies computational problems according to their inherent difficulty. A computational problem can be viewed as an infinite collection of instances together with a solution for every instance. It is conventional to represent both instances and solutions by binary strings. We adopt $\mathbb{B} = \{\mathtt{t}, \mathtt{f}\}$ as the preferred binary alphabet and associate with each computational problem a partial function $F : \mathbb{B}^* \xrightarrow{p} \mathbb{B}$ deciding partially whether a certain instance has a solution. In this section, we study the complexity of computing such computational problems. In the sequel, we denote by $\mathcal{IS}^{lf}(A)$ the set of *loop-free* $\mathrm{PGLB}_{bt}(A)$ instruction sequences, i.e. the set of $\mathrm{PGLB}_{bt}(A)$ instruction sequences without backward jumps. Moreover, we write $\mathit{length}(I)$ for the number of instructions of $I \in \mathcal{IS}(A)$.

**Definition 4.1.**

1. *Let $\mathcal{F} = \mathcal{F}_{\mathtt{in}} \cup \mathcal{F}_{\mathtt{aux}}$ where $\mathcal{F}_{\mathtt{in}} = \{\mathtt{in}{:}n \mid n \in \mathbb{N}\}$ and $\mathcal{F}_{\mathtt{aux}} = \{\mathtt{aux}{:}n \mid n \in \mathbb{N}\}$, $\mathcal{M} = \{\mathtt{set{:}t}, \mathtt{set{:}f}, \mathtt{get}\}$ and $A = \{f.m \mid f \in \mathcal{F}, m \in \mathcal{M}\}$.*

2. *Let $f_1, \ldots, f_n \in \mathcal{F}$ and $\mathbb{S}_1, \ldots, \mathbb{S}_n$ be services. Then $\oplus_{i=1}^{n} f_i.\mathbb{S}_i$ denotes the service family $f_1.\mathbb{S}_1 \oplus \cdots \oplus f_n.\mathbb{S}_n$.*

3. *Let $F : \mathbb{B}^k \xrightarrow{p} \mathbb{B}$ be a $k$-ary partial function on the Booleans $\mathbb{B}$. $I \in \mathcal{IS}(A)$ is said to* compute $F$ using $l$ auxiliary registers *if for all $b_1, \ldots, b_k \in \mathbb{B}$*

$$(|I|/ \oplus_{i=1}^{l} \mathtt{aux}{:}i.B(\mathtt{t}))!\, \oplus_{i=1}^{k} \mathtt{in}{:}i.B(b_i) = \begin{cases} F(b_1, \ldots, b_k) & \text{if } F(b_1, \ldots, b_k) \text{ is defined,} \\ \mathtt{d} & \text{otherwise.} \end{cases}$$

*Moreover, we say that $I$* computes *$F$ if $I$ computes $F$ using $l$ auxiliary registers for some $l \in \mathbb{N}$, and $I$* computes *$F$* without the use of auxiliary registers *if $l = 0$.*

**Proposition 4.2.** *Let $F : \mathbb{B}^k \xrightarrow{p} \mathbb{B}$ be a $k$-ary partial function on the Booleans $\mathbb{B}$. Then $F$ can be computed by an $I \in \mathcal{IS}^{lf}(A)$ with length $3 \times 2^k - 2$ without the use of auxiliary registers.*

**Proof:** By induction on $k$, we construct an instruction sequence $I_F \in \mathcal{IS}^{lf}(A)$ that computes $F$. If $k = 0$, then $F()$ is either $\mathtt{t}$, $\mathtt{f}$ or undefined. Thus we can take for $I_F$ either $!\mathtt{t}$, $!\mathtt{f}$ or $\#0$. Let $F$ be $k+1$-ary and consider the functions $G_b(b_1, \ldots, b_k) = F(b_1, \ldots, b_k, b)$ with $b \in \{\mathtt{t}, \mathtt{f}\}$. By the induction hypothesis $G_b$ can be computed by some $I_{G_b} \in \mathcal{IS}^{lf}(A)$

with length $3 \times 2^k - 2$ without the use of auxiliary registers. Then

$$(|-\mathtt{in}{:}k{+}1.\mathtt{get}; \#3{\times}2^k{-}1; I_{G_\mathtt{t}}; I_{G_\mathtt{f}}|/\emptyset)!{\oplus}_{i=1}^{k+1} i.B(b_i) = \begin{cases} (|I_{G_\mathtt{t}}|/\emptyset)! \oplus_{i=1}^{k} i.B(b_i) & \text{if } b_{k+1} = \mathtt{t}, \\ (|I_{G_\mathtt{f}}|/\emptyset)! \oplus_{i=1}^{k} i.B(b_i) & \text{if } b_{k+1} = \mathtt{f}, \\ \mathtt{d} & \text{otherwise.} \end{cases}$$

Thus $I_F = -\mathtt{in}{:}k+1.\mathtt{get}; \#3 \times 2^k - 1; I_{G_\mathtt{t}}; I_{G_\mathtt{f}}$ computes $F$ without the use of auxiliary registers or backward jumps and has length $2 + 2 \times (3 \times 2^k - 2) = 3 \times 2^{k+1} - 2$. □

Thus backward jumps are not necessary for the computation of partial Boolean functions. However, they can make a contribution to the expressiveness of $\mathrm{PGLB}_{bt}(A)$ by allowing shorter instruction sequences for computing a given computational problem.

**Definition 4.3.** *For $F : \mathbb{B}^* \xrightarrow{p} \mathbb{B}$, we denote by $F_k$ ($k \in \mathbb{N}$) the restriction of $F$ to $\mathbb{B}^k$ and distinguish the following three classes of computational problems.*

1. $\mathcal{IS}_P^{lf}(A) =$

$$\{F : \mathbb{B}^* \xrightarrow{p} \mathbb{B} \quad | \quad \begin{array}{l} \textit{there exists a polynomial function } h : \mathbb{N} \to \mathbb{N} \\ \textit{such that for all } k \in \mathbb{N}, \\ F_k \textit{ can be computed by an } I \in \mathcal{IS}^{lf}(A) \textit{ with } \mathit{length}(I) \leq h(k)\} \end{array}$$

2. $\mathcal{IS}_P(A) =$

$$\{F : \mathbb{B}^* \xrightarrow{p} \mathbb{B} \quad | \quad \begin{array}{l} \textit{there exists a polynomial function } h : \mathbb{N} \to \mathbb{N} \\ \textit{such that for all } k \in \mathbb{N}, \\ F_k \textit{ can be computed by an } I \in \mathcal{IS}(A) \textit{ with } \mathit{length}(I) \leq h(k)\} \end{array}$$

3. $\mathcal{IS}_E^{lf}(A) =$

$$\{F : \mathbb{B}^* \xrightarrow{p} \mathbb{B} \quad | \quad \begin{array}{l} \textit{there exists a } c \in \mathbb{N} \textit{ such that for all } k \in \mathbb{N}, \\ F_k \textit{ can be computed by an } I \in \mathcal{IS}^{lf}(A) \textit{ with } \mathit{length}(I) \leq c \times 2^k\} \end{array}$$

In the sequel we denote by $[\mathbb{B}^* \xrightarrow{p} \mathbb{B}]$ the set of all partial functions from $\mathbb{B}^*$ to $\mathbb{B}$, and by $[\mathbb{B}^* \longrightarrow \mathbb{B}]$ the set of all total functions from $\mathbb{B}^*$ to $\mathbb{B}$. Restating Proposition 4.2, we have

**Proposition 4.4.** $\mathcal{IS}_E^{lf}(A) = [\mathbb{B}^* \xrightarrow{p} \mathbb{B}]$

In nonuniform complexity theory, P/poly is the complexity class of computational problems solved by a polynomial-time Turing machine with a polynomial-bounded advice function. It is also equivalently defined as the class PSIZE of problems that have polynomial-size Boolean circuits.

**Theorem 4.5.** $\mathcal{IS}_P^{lf}(A) \cap [\mathbb{B}^* \longrightarrow \mathbb{B}] = \mathrm{P/poly}$

**Proof:** We shall prove the inclusion $\subseteq$ using the definition of P/poly in terms of Turing machines that take advice, and the inclusion $\supseteq$ using the definition in terms of Boolean circuits.

$\subseteq$: Suppose that $F \in \mathcal{IS}_P^{lf}(A) \cap [\mathbb{B}^* \to \mathbb{B}]$. Then, for all $k \in \mathbb{N}$, there exists an $I_k \in \mathcal{IS}^{lf}(A)$ that computes $F_k$ with $\mathit{length}(I_k)$ polynomial in $k$. Then $F$ can be computed by a Turing machine that, on input of size $k$, takes a binary description of $I_k$ as advice and then just simulates the execution of $I_k$. It is easy to see that under the assumption that instructions of the form $\mathtt{in}{:}i.m, +\mathtt{in}{:}i.m, -\mathtt{in}{:}i.m$ with $i > k$, and $\mathtt{aux}{:}i.m, +\mathtt{aux}{:}i.m, -\mathtt{aux}{:}i.m$, and $\#i$ with $i > \mathit{length}(I_k)$ do not occur in $I_k$, the size of the description of $I_k$ and the number of steps that it takes to simulate its execution are both polynomial in $k$. It is obvious that we can make the assumption without loss of generality. Hence, $F$ is also in P/poly.

$\supseteq$: We first show that a function $F : \mathbb{B}^k \to \mathbb{B}$ that is induced by a Boolean circuit $C$ consisting of NOT, AND and OR gates can be computed by an $I_C \in \mathcal{IS}^{lf}(A)$. More precisely, assuming that $\{g_{i_1}, \ldots, g_{i_n}\}$ $(i_1, \ldots, i_n \in \mathbb{N})$ is a topological ordering of the gates with output node $g_{i_n}$, we prove by induction on $n$ that we may assume that $I_C$ is of the form $I; +\mathtt{aux}{:}i_n.\mathtt{get}; \mathtt{!t}; \mathtt{!f}$ for some $I \in \mathcal{IS}^{lf}(A)$ with $\mathit{length}(I) \leq 4 \times n$.

If $n = 1$, then depending on the form of the single gate either

$$\begin{array}{lcl} I_\neg & = & +\mathtt{in}{:}i.\mathtt{get}; \mathtt{aux}{:}i_1.\mathtt{set{:}f}; +\mathtt{aux}{:}i_1.\mathtt{get}; \mathtt{!t}; \mathtt{!f}, \\ I_\wedge & = & -\mathtt{in}{:}i.\mathtt{get}; \#2; -\mathtt{in}{:}j.\mathtt{get}; \mathtt{aux}{:}i_1.\mathtt{set{:}f}; +\mathtt{aux}{:}i_1.\mathtt{get}; \mathtt{!t}; \mathtt{!f}, \text{ or} \\ I_\vee & = & +\mathtt{in}{:}i.\mathtt{get}; \#3; -\mathtt{in}{:}j.\mathtt{get}; \mathtt{aux}{:}i_1.\mathtt{set{:}f}; +\mathtt{aux}{:}i_1.\mathtt{get}; \mathtt{!t}; \mathtt{!f} \end{array}$$

with properly chosen $i, j$ comply. For the induction step we again have to distinguish three cases. We here consider only the case that $g_{i_n}$ is an $AND$ gate. Suppose that the input of $g_{i_n}$ are the output gates $g_{i_l}$ and $g_{i_m}$ of the subcircuits $C'$ and $C''$. By the induction hypothesis we may assume that the functions induced by $C'$ and $C''$ can be computed by the $\mathcal{IS}^{lf}(A)$ instruction sequences $I_{C'} = I'; +\mathtt{aux}{:}i_l.\mathtt{get}; \mathtt{!t}; \mathtt{!f}$ and $I_{C''} = I''; +\mathtt{aux}{:}i_m.\mathtt{get}; \mathtt{!t}; \mathtt{!f}$ with $\mathit{length}(I_{C'}) \leq 4 \times |C'|$ and $\mathit{length}(I_{C''}) \leq 4 \times |C''|$ where the sizes $|C'|$ and $|C''|$ are the number of gates in the respective subcircuits. Then

$$I_C = I'; I''; -\mathtt{aux}{:}i_l.\mathtt{get}; \#2; -\mathtt{aux}{:}i_m.\mathtt{get}; \mathtt{aux}{:}i_n.\mathtt{set{:}f}; +\mathtt{aux}{:}i_n.\mathtt{get}; \mathtt{!t}; \mathtt{!f}$$

computes $F$ and $\mathit{length}(I) = \mathit{length}(I') + \mathit{length}(I'') \leq 4 \times |C'| + 4 \times |C'| \leq 4 \times n$. If one input is an input node, a shorter instruction sequence suffices, e.g. $I'; -\mathtt{in}{:}j.\mathtt{get}; \#2; -\mathtt{aux}{:}i_l.\mathtt{get}; \mathtt{aux}{:}i_n.\mathtt{set{:}f}; +\mathtt{aux}{:}i_n.\mathtt{get}; \mathtt{!t}; \mathtt{!f}$.

Now suppose that $F \in$ P/poly. Then, for all $k \in \mathbb{N}$, there exists a Boolean circuit $C_k$ such that $C_k$ computes $F_k$ and the size of $C_k$ is polynomial in $k$. From the above and the fact that linear in the size of $C_k$ implies polynomial in $k$, it follows that $F$ is also in $\mathcal{IS}_P^{lf}(A)$. □

Combining Proposition 4.4 and the previous theorem, we have

**Corollary 4.6.** P/poly $\subsetneq \mathcal{IS}_P^{lf}(A) \subseteq \mathcal{IS}_P(A) \subseteq \mathcal{IS}_E^{lf}(A)$

In the remainder of this section we shall show—adopting a reasonable assumption—that also the second inclusion is proper.

The satisfiability problem $3SAT$ is concerned with efficiently finding a satisfying assignment to a propositional formula. The input is a conjunctive normal form where each clause is limited to at most 3 literals—a *3-CNF formula*. The goal is to find an assignment to the variables that makes the entire expression true, or to prove that no such assignment exists. This problem is NP-complete, and therefore no polynomial-time algorithm can succeed on all 3-CNF formulae unless $NP \subseteq \mathrm{P/poly}$ [12, 14]. The latter implies the collapse of the polynomial hierachy as was proved by Karp and Lipton in 1980 [13].

$3SAT(k)$ can be computed by instruction sequences with polynomial length if we allow backward jumps. Under the hypothesis that $NP \not\subseteq \mathrm{P/poly}$, it then follows that instruction sequences for this decision problem without backward jumps have to be significantly longer.

**Theorem 4.7.** $3SAT \in \mathcal{IS}_P(A)$

**Proof:** If the number of Boolean variables is $k$, then there are $8k^3$ possible clauses of length 3—we allow multiple occurrences of a variable in a single clause and neglect the order of the literals. We will encode a 3-CNF $\psi$ over $k$ Boolean variables as a sequence of Boolean values $\langle b\rangle_\psi$ of length $8k^3$ where a `t` indicates that a certain clause occurs in the 3-CNF and a `f` excludes the clause. Vice versa, given a sequence $\langle b\rangle \in \mathbb{B}^{8k^3}$ we denote the 3-CNF obtained from $\langle b\rangle$ by $\psi_{\langle b\rangle}$ and define $3SAT(k) : \mathbb{B}^{8k^3} \to \mathbb{B}$ by

$$3SAT(k)(\langle b\rangle) = \begin{cases} \mathtt{t} & \text{if } \psi_{\langle b\rangle} \text{ is satisfiable,} \\ \mathtt{f} & \text{otherwise.} \end{cases}$$

We let $\{v_1, \ldots, v_k\}$ be Boolean variables and define for $\langle l, m, n, i\rangle \in \{1, \ldots, k\}^3 \times \{1, \ldots, 8\}$ the clause $\gamma_{\langle l,m,n,i\rangle}$ by

$$\gamma_{\langle l,m,n,i\rangle} = \begin{cases} v_l \vee v_m \vee v_n & \text{if } i = 1, \\ v_l \vee v_m \vee \neg v_n & \text{if } i = 2, \\ v_l \vee \neg v_m \vee v_n & \text{if } i = 3, \\ v_l \vee \neg v_m \vee \neg v_n & \text{if } i = 4, \\ \neg v_l \vee v_m \vee v_n & \text{if } i = 5, \\ \neg v_l \vee v_m \vee \neg v_n & \text{if } i = 6, \\ \neg v_l \vee \neg v_m \vee v_n & \text{if } i = 7, \\ \neg v_l \vee \neg v_m \vee \neg v_n & \text{if } i = 8, \end{cases}$$

and the instruction sequence $CHECK_{\langle l,m,n,i\rangle}$ by

$$CHECK_{\langle l,m,n,i\rangle} = \begin{cases} +\mathtt{aux}{:}l.\mathtt{get}; \#2; +\mathtt{aux}{:}m.\mathtt{get}; \#2; +\mathtt{aux}{:}n.\mathtt{get} & \text{if } i = 1, \\ +\mathtt{aux}{:}l.\mathtt{get}; \#2; +\mathtt{aux}{:}m.\mathtt{get}; \#2; -\mathtt{aux}{:}n.\mathtt{get} & \text{if } i = 2, \\ +\mathtt{aux}{:}l.\mathtt{get}; \#2; -\mathtt{aux}{:}m.\mathtt{get}; \#2; +\mathtt{aux}{:}n.\mathtt{get} & \text{if } i = 3, \\ +\mathtt{aux}{:}l.\mathtt{get}; \#2; -\mathtt{aux}{:}m.\mathtt{get}; \#2; -\mathtt{aux}{:}n.\mathtt{get} & \text{if } i = 4, \\ -\mathtt{aux}{:}l.\mathtt{get}; \#2; +\mathtt{aux}{:}m.\mathtt{get}; \#2; +\mathtt{aux}{:}n.\mathtt{get} & \text{if } i = 5, \\ -\mathtt{aux}{:}l.\mathtt{get}; \#2; +\mathtt{aux}{:}m.\mathtt{get}; \#2; -\mathtt{aux}{:}n.\mathtt{get} & \text{if } i = 6, \\ -\mathtt{aux}{:}l.\mathtt{get}; \#2; -\mathtt{aux}{:}m.\mathtt{get}; \#2; +\mathtt{aux}{:}n.\mathtt{get} & \text{if } i = 7, \\ -\mathtt{aux}{:}l.\mathtt{get}; \#2; -\mathtt{aux}{:}m.\mathtt{get}; \#2; -\mathtt{aux}{:}n.\mathtt{get} & \text{if } i = 8. \end{cases}$$

Observe that the snippet $CHECK_{\langle l,m,n,i\rangle}$ checks whether a certain assignment—held in the auxiliary registers $b_1, \ldots, b_k$—satisfies clause $\gamma_{\langle l,m,n,i\rangle}$. We fix an arbitrary bijection $\phi : \{1, \ldots, 8n^3\} \to \{1, \ldots, n\}^3 \times \{1, \ldots, 8\}$ and put

$$m \to CHECK_{\phi(m)} = -\texttt{in:}m.\texttt{get}; \#8; CHECK_{\phi(m)}; \#2; \#9$$

if $1 \leq m < 8k^3$, and

$$8k^3 \to CHECK_{\phi(8k^3)} = -\texttt{in:}8k^3.\texttt{get}; \#6; CHECK_{\phi(8k^3)}; !\texttt{t}$$

and join the conditional checks to form the instruction sequence

$$CHECK = 1 \to CHECK_{\phi(1)}; \ldots; 8k^3 \to CHECK_{\phi(8k^3)}.$$

Thus, if $\gamma_{\phi(m)}$ is a clause of the 3-CNF that is satisfied by the current assignment, or if the clause does not occur in the 3-CNF, then execution of $CHECK$ continues after the snippet $m \to CHECK_m$ with the snippet $m+1 \to CHECK_{m+1}$ if $m < 8k^3$ and terminates with reply $\texttt{t}$ if $m = 8k^3$. If, however, $\gamma_{\phi(m)}$ is not satisfied by the assignment then execution jumps with chained jumps of length 9 or—if $m = 8k^3$—a single jump of length 6 to the first instruction after $CHECK$.

In order to generate assignments we use the snippet

$$NEXT = NEXT_1; \cdots; NEXT_k$$

where

$$NEXT_i = -\texttt{aux:}i.\texttt{get}; \#3; \texttt{aux:}i.\texttt{set:f}; \#5; \texttt{aux:}i.\texttt{set:t}$$

for $1 \leq i < k$ and

$$NEXT_k = -\texttt{aux:}k.\texttt{get}; \#3; \texttt{aux:}k.\texttt{set:f}; \#3; \texttt{aux:}k.\texttt{set:t}; !\texttt{f}$$

Observe that, starting with the assignment $\oplus_{i=1}^{k} \texttt{aux:}i.B(\texttt{t})$, repeated execution of $NEXT$ generates all possible assignments. After the last assignment $\oplus_{i=1}^{k} \texttt{aux:}i.B(\texttt{f})$, all values are set back to $\texttt{t}$ and the generator terminates with reply $\texttt{f}$.

We combine the checks and the assignment generator and define

$$I_k = CHECK; NEXT; \backslash\#(72k^3 + 5k).$$

Then

$$(|I_k|/ \oplus_{i=1}^{k} \texttt{aux:}i.B(\texttt{t}))! \oplus_{i=1}^{8k^3} \texttt{in:}i.B(b_i) = \begin{cases} \texttt{t} & \text{if } \psi_{\langle b_i \rangle} \text{ is satisfiable,} \\ \texttt{f} & \text{otherwise.} \end{cases}$$

Thus $I_k$ computes $3SAT(k)$ using $k$ auxiliary registers. Since $I_k$ has $72k^3+5k+1$ instructions, we may conclude that $3SAT \in \mathcal{IS}_P(A)$. □

**Corollary 4.8.** *If* $NP \not\subseteq$ P/poly *then* $\mathcal{IS}_P^{lf}(A) \subsetneq \mathcal{IS}_P(A)$.

**Proof:** Suppose $3SAT \in \mathcal{IS}_P^{lf}$, then $3SAT \in$ P/poly by Theorem 4.5 and hence $NP \subseteq$ P/poly. □

## 5 Conclusion

Program algebra is a setting suited for investigating instruction sequences. In this setting, we have shown that each partial Boolean function can be computed by an instruction sequence without the use of auxiliary registers or backward jumps. Hence backward jumps do not contribute to the expressiveness of instruction sequences. However, instruction sequences can be significantly shorter when backward jumps and auxiliary registers are permitted. Thus, semantically we can do without backward jumps. However, if we want to avoid an explosion of the length of instruction sequences, then backward jumps are essential. It remains an open problem whether the third inclusion in Corollary 4.6 is proper.

## References


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

[2] J.A. Bergstra, I. Bethke, and A. Ponse. Decision problems for pushdown threads. *Acta Informatica*, 44(2):75–90, 2007.

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

[4] J.A. Bergstra and C.A. Middelburg. Thread algebra for strategic interleaving. *Formal Aspects of Computing*, 19(4):445–474, 2007.

[5] J.A. Bergstra and C.A. Middelburg. *Functional units for natural numbers*. arXiv:0911.1851v1, 2009.

[6] J.A. Bergstra and C.A. Middelburg. *Instruction sequence processing operators*. arXiv:0909.2088, 2009.

[7] J.A. Bergstra and C.A. Middelburg. *Autosolvability of Halting Problem instances for instruction sequences*. arXiv:0911.5018, 2009.

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

[9] D.B. Bui and A.V. Mavlyanov Theory of program algebras. *Ukrainian Mathmatical Journal*, 36(6):761–764, 1984.

[10] D.B. Bui and A.V. Mavlyanov Mutual derivability of operations in program algebra. I *Cybernetics and Systems Analysis*, 24(1):35–39, 1988.

[11] D.B. Bui and A.V. Mavlyanov Mutual derivability of operations in program algebra. II *Cybernetics and Systems Analysis*, 24(6):1–6, 1988.

[12] S. Cook. The complexity of theorem-proving procedures. In *Proc. 3rd FOCS*, IEEE Computer Society, 151–158, 1971.

[13] R.M. Karp and R.J. Lipton. Some connections between nonuniform and uniform complexity classes. In *Proc. 12th STOC*, ACM, 302–309, 1980.

[14] L.A. Levin. Universal enumeration problems. *Problemy Peredači Informacii* 9(3):115–116, 1973.

[15] A. Ponse and M.B. van der Zwaag. An introduction to program and thread algebra. In A. Beckmann et al. (editors), *Logical Approaches to Computational Barriers: Proceedings CiE 2006*, LNCS 3988, pages 445-458, Springer-Verlag, 2006.

[16] J. von Wright. *An Interactive Metatool for Exploring Program Algebras.* Turku Centre for Computer Science, TUCS Technical Report No. 247, March, 1999.