Theory Comput Syst (2012) 50:706–720
DOI 10.1007/s00224-011-9376-x

On the Contribution of Backward Jumps to Instruction Sequence Expressiveness

Jan A. Bergstra · Inge Bethke

Published online: 25 November 2011
© The Author(s) 2011. This article is published with open access at Springerlink.com

Abstract We investigate the expressiveness of backward jumps in a frame work of formalized sequential programming called *program algebra* and characterize established non-uniform complexity classes in terms of instruction sequences, backward jumps and auxiliary registers.

Keywords Program algebra · Instruction sequences · Backward jumps · Non-uniform complexity

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. [4, 9–11, 22]). Here the notion of program algebra refers to the concept introduced in [4] 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

J.A. Bergstra · I. Bethke (✉)
Section Theory of Computer Science, Informatics Institute, University of Amsterdam, Amsterdam, The Netherlands
e-mail: I.Bethke@uva.nl
url: www.science.uva.nl/~inge

J.A. Bergstra
url: www.science.uva.nl/~janb

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 infinite threads 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 functions that can be computed with the aid of instruction sequences. As it will turn out every 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 of instruction sequences, then backward jumps are essential.

This paper is organized as follows. Section 2 briefly recalls the program notation $\mathrm{PGLB}_{\mathrm{bt}}$, its accompanying thread algebra and the interactions of services with threads. Section 3 introduces a hierarchy of decision problems that can be computed non-uniformly by instruction sequences of bounded length with restricted use of services. Allowing or disallowing backward jumps leads to the characterization of established complexity classes.

2 Instruction Sequences, Regular Threads and Services

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 [4] (see also [17]). $\mathrm{PGLB}_{\mathrm{bt}}$ is PGLB with the termination instruction ! refined into two Boolean termination instructions $\texttt{!t}, \texttt{!f}$ (see also [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 $\texttt{a}, \texttt{b}, \texttt{c}, \ldots$. $\mathrm{PGLB}_{\mathrm{bt}}(A)$ *instruction sequences* are then of the following form ($\texttt{a} \in A$, $l \in \mathbb{N}$):

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

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

1. *Basic instructions* $\texttt{a}, \texttt{b}, \texttt{c}, \ldots$ prescribe actions that are considered indivisible and executable in finite time, and return upon execution a Boolean reply value $\texttt{t}$ or $\texttt{f}$ that may be used for subsequent program control.

2.–3. *Test instructions* obtained from basic instructions $\texttt{a} \in A$ by prefixing them with either $+$ (positive test) or $-$ (negative test) control subsequent execution via the reply of their execution as follows. When a positive test is performed, the basic instruction is executed and, in case $\texttt{t}$ is returned, the remaining sequence of instructions. If there are no remaining instructions, inaction occurs. In the case that $\texttt{f}$ is returned, the next instruction is skipped and execution proceeds with the instruction following the skipped one. If no such instruction exists, inaction occurs. Execution of a negative test is the same, except that the roles of $\texttt{t}$ and $\texttt{f}$ are interchanged.

4.–5. *Jump instructions* $\#l, \backslash\#l$ prescribe to jump l instructions forward and backward, respectively—if possible; otherwise inaction occurs. In particular, #0 and \#0 jump to itself and inaction occurs.

6.–7. *Termination instructions* $!\mathtt{t}, !\mathtt{f}$ yield termination and 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. [2, 4] under the name Polarized Process Algebra.

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

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

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 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 $\{E_i = T_i \mid i \in I\}$ where each T_i is of the form $\mathsf{S+}, \mathsf{S-}, \mathsf{D}$ or $T \trianglelefteq \mathtt{a} \trianglerighteq T'$ with T, T' thread expressions in which variables from $\{E_i \mid i \in I\}$ may occur. A *regular* thread is the solution of a finite guarded recursive specification, i.e. a guarded recursive specification with a finite number of recursive equations.

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, inaction occurs. For the execution of a negative test instruction $-\mathtt{a}$, subsequent execution is prescribed by the complementary replies. If we add the rule

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

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

Table 1 Equations for thread extraction, where $X = u_1; \ldots; u_k$, a ranges over basic instructions and $i, k, l \in \mathbb{N}$

$\lvert i, X\rvert = \mathsf{D}$	if $i = 0$ or $k < i$
$\lvert i, X\rvert = \mathtt{a} \circ \lvert i+1, X\rvert$	if $u_i = \mathtt{a}$
$\lvert i, X\rvert = \lvert i+1, X\rvert \trianglelefteq \mathtt{a} \trianglerighteq \lvert i+2, X\rvert$	if $u_i = +\mathtt{a}$
$\lvert i, X\rvert = \lvert i+2, X\rvert \trianglelefteq \mathtt{a} \trianglerighteq \lvert i+1, X\rvert$	if $u_i = -\mathtt{a}$
$\lvert i, X\rvert = \lvert i+l, X\rvert$	if $u_i = \#l$
$\lvert i, X\rvert = \lvert i-l, X\rvert$	if $u_i = \backslash\#l$ and $i > l$
$\lvert i, X\rvert = \mathsf{D}$	if $u_i = \backslash\#l$ and $i \leq l$
$\lvert i, X\rvert = \mathsf{S+}$	if $u_i = !\mathtt{t}$
$\lvert i, X\rvert = \mathsf{S-}$	if $u_i = !\mathtt{f}$

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

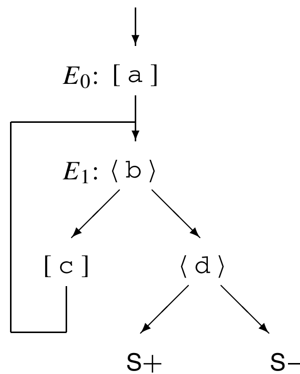
$$X = \mathtt{a}; +\mathtt{b}; \#2; \#3; \mathtt{c}; \backslash\#4; +\mathtt{d}; !\mathtt{t}; !\mathtt{f}.$$

Thread extraction of X yields the regular thread

$$E_0 = \mathtt{a} \circ E_1$$

$$E_1 = \mathtt{c} \circ E_1 \trianglelefteq \mathtt{b} \trianglerighteq (\mathsf{S+} \trianglelefteq \mathtt{d} \trianglerighteq \mathsf{S-})$$

A picture of this thread is



Here [a] corresponds to action prefix and ⟨a⟩ 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.

For basic information on thread algebra we refer to [3, 17]; more advanced matters, such as an operational semantics for thread algebra, are discussed in [5].

Services were first introduced as *state machines* in [8]. They are Mealy machines [16] which support a thread in its execution—for example as memory device—and in doing so produce replies and undergo possible changes. 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* $\mathbb{S} = \langle S, \mathit{eff}, \mathit{yld}, s_0\rangle$ consists of

1. A set S of *states*.
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* $s_0 \in S$.

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 $\mathbb{S}(m) = \mathit{yld}(m, s_0)$.

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 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.

Example 2.2 Given the set of methods $\mathcal{M} = \{\mathtt{set{:}t}, \mathtt{set{:}f}, \mathtt{get}\}$, we consider the service $B(x)$ of a Boolean register with initial value $x \in \mathcal{R}$: $B(x) = \langle \mathcal{R}, \mathit{eff}, \mathit{yld}, x\rangle$ where

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

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

and $\mathit{eff}(\mathtt{get}, x) = \mathit{yld}(\mathtt{get}, x) = x$. Observe that

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

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

Services model part of an execution environment in which a thread may make use of services by requesting a service to process a method and to return a reply value at completion. We combine threads with services and extend the combination with the two operators / (*use*) and ! (*reply*) which relate to this kind of interaction. An axiomatization for the use and reply operator was first given in [7]. Here we will only consider the environment in which a thread can reply to Boolean read-only input registers using auxiliary Boolean registers. In this setting use and reply can be defined as follows.

Definition 2.3 Let $\mathcal{M} = \{\texttt{set:t}, \texttt{set:f}, \texttt{get}\}$, $A = \{\texttt{aux:}i.m \mid m \in \mathcal{M}, i \in \mathbb{N}\} \cup \{\texttt{in:}i.\texttt{get} \mid i \in \mathbb{N}\}$ and $\mathcal{B}, \mathcal{B}' \subseteq \{B_i \mid i \in \mathbb{N}\}$ be disjoint sets of Boolean auxiliary and input registers. For a regular thread T, $(T/\mathcal{B})!\mathcal{B}'$ is defined inductively by

1. if $T = \mathsf{S+}$, then $(T/\mathcal{B})!\mathcal{B}' = \texttt{t}$,
2. if $T = \mathsf{S-}$, then $(T/\mathcal{B})!\mathcal{B}' = \texttt{f}$,
3. if $T = \mathsf{D}$, then $(T/\mathcal{B})!\mathcal{B}' = \texttt{d}$,
4. if $T = T_1 \trianglelefteq \texttt{aux:}i.m \trianglerighteq T_2$, then

$$(T/\mathcal{B})!\mathcal{B}' = \begin{cases} (T_1/((\mathcal{B} - \{B_i\}) \cup \{\frac{\partial}{\partial m} B_i\}))!\mathcal{B}' & \text{if } B_i \in \mathcal{B} \text{ and } B_i(m) = \texttt{t} \\ (T_2/((\mathcal{B} - \{B_i\}) \cup \{\frac{\partial}{\partial m} B_i\}))!\mathcal{B}' & \text{if } B_i \in \mathcal{B} \text{ and } B_i(m) = \texttt{f} \\ \texttt{d} & \text{if } B_i \in \mathcal{B} \text{ and } B_i(m) = \texttt{d} \\ (T_1/\mathcal{B})!\mathcal{B}' \trianglelefteq \texttt{aux:}i.m \trianglerighteq (T_2/\mathcal{B})!\mathcal{B}' & \text{otherwise,} \end{cases}$$

5. if $T = T_1 \trianglelefteq \texttt{in:}i.\texttt{get} \trianglerighteq T_2$, then

$$(T/\mathcal{B})!\mathcal{B}' = \begin{cases} (T_1/\mathcal{B})!\mathcal{B}' & \text{if } B_i \in \mathcal{B}' \text{ and } B_i(m) = \texttt{t} \\ (T_2/\mathcal{B})!\mathcal{B}' & \text{if } B_i \in \mathcal{B}' \text{ and } B_i(m) = \texttt{f} \\ \texttt{d} & \text{otherwise.} \end{cases}$$

Here we assume that methods that cannot be processed because of a shortage of auxiliary registers remain unprocessed whereas a reply to not existing input variables results in divergence.

Example 2.4 We continue with Example 2.2 and let $Eq(1,2)$ be the $\text{PGLB}_{\text{bt}}(A)$ instruction sequence

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

which intuitively describes a thread that compares 2 input registers and returns the reply t if their values are not divergent and equal, f if their values are not divergent but different, and d otherwise. Indeed, formalizing this interaction in the setting of services and threads we put $\mathcal{B} = \{B_1(b_1), B_2(b_2)\}$ and compute

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

The use of auxiliary registers can be illustrated as follows. We let

$$Eq(1,2,3) = +\texttt{in:1.get}; \#2; \#4; -\texttt{in:2.get}; !\texttt{f}; \#4;$$

$$+\,\texttt{in:2.get};\,\backslash\#3;\,\texttt{aux:0.set:f};\,I$$

$$I = +\texttt{in:3.get};\,\#2;\,\#4;\,+\texttt{aux:0.get};\,!\texttt{t};\,!\texttt{f};\,-\texttt{aux:0.get};\,\backslash\#3;\,\backslash\#3$$

be the lazy equality test of 3 registers which stores an intermediate result in the auxiliary register $B_0(\texttt{t})$. Assuming $\mathcal{B} = \{B_1(b_1), B_2(b_2), B_3(b_3)\}$ we have

$$
\begin{aligned}
&(|Eq(1,2,3)|/\{B_0(\texttt{t})\})!\mathcal{B}\\
&\quad = (((|I| \trianglelefteq \texttt{in:2.get} \trianglerighteq \mathsf{S}-) \trianglelefteq \texttt{in:1.get} \trianglerighteq\\
&\qquad (\mathsf{S}- \trianglelefteq \texttt{in:2.get} \trianglerighteq (\texttt{aux:0.set:f} \circ |I|)))/B_0(\texttt{t}))!\mathcal{B}\\
&\quad = \begin{cases} ((|I| \trianglelefteq \texttt{in:2.get} \trianglerighteq \mathsf{S}-)/\{B_0(\texttt{t})\})!\mathcal{B} & \text{if } b_1 = \texttt{t},\\ ((\mathsf{S}- \trianglelefteq \texttt{in:2.get} \trianglerighteq (\texttt{aux:0.set:f} \circ |I|))/\{B_0(\texttt{t})\})!\mathcal{B} & \text{if } b_1 = \texttt{f},\\ \texttt{d} & \text{if } b_1 = \texttt{d}, \end{cases}\\
&\quad = \begin{cases} (|I|/\{B_0(\texttt{t})\})!\mathcal{B} & \text{if } b_1 = \texttt{t} = b_2,\\ ((\texttt{aux:0.set:f} \circ |I|/\{B_0(\texttt{t})\})!\mathcal{B} & \text{if } b_1 = \texttt{f} = b_2,\\ \texttt{f} & \text{if } \texttt{d} \neq b_1 \neq b_2 \neq \texttt{d},\\ \texttt{d} & \text{if } b_1 = \texttt{d} \text{ or } b_2 = \texttt{d}, \end{cases}\\
&\quad = \begin{cases} (|I|/\{B_0(\texttt{t})\})!\mathcal{B} & \text{if } b_1 = \texttt{t} = b_2,\\ (|I|/\{B_0(\texttt{f})\})\mathcal{B} & \text{if } b_1 = \texttt{f} = b_2,\\ \texttt{f} & \text{if } \texttt{d} \neq b_1 \neq b_2 \neq \texttt{d},\\ \texttt{d} & \text{if } b_1 = \texttt{d} \text{ or } b_2 = \texttt{d}, \end{cases}
\end{aligned}
$$

Thus

$$(|Eq(1,2,3)|/\{B_0(\texttt{t})\})!\mathcal{B} = \begin{cases} \texttt{t} & \text{if } b_1 = b_2 = b_3 \neq \texttt{d},\\ \texttt{d} & \text{if } b_1 = \texttt{d} \text{ or } b_2 = \texttt{d} \text{ or } b_1 = b_2 \neq \texttt{d} = b_3,\\ \texttt{f} & \text{otherwise.} \end{cases}$$

Equality tests for 3 registers can be written in several ways: e.g. without backward jumps by replacing the jump \#3 by !f. Also the use of an auxiliary register can be omitted. We shall come back to this issue in Proposition 3.2.

For more information about services we refer to [7, 8, 17].

3 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

$$+\texttt{aux:1.get};\,\#2;\,\#3;\,\texttt{aux:1.set:f};\,\backslash\#4;\,\texttt{aux:1.set:t};\,\backslash\#2$$

a $\mathrm{PGLB_{bt}}(A)$ instruction sequence which prescribes the repeated swap of a register content. Clearly no $X \in \mathcal{IS}(A)$ without backward jumps can produce a thread with

an unbounded number of successive swaps. Thus backward jumps add to the expressiveness of $\mathrm{PGLB}_{\mathrm{bt}}(A)$.

In the field of computational complexity theory one classifies decidable decision problems according to their inherent difficulty. A decision problem can be viewed as an infinite collection of instances each of which can be answered by either yes or no. It is conventional to represent the instances by binary strings. We adopt $\mathbb{B} = \{\mathtt{t}, \mathtt{f}\}$ as the preferred binary alphabet and associate with each decision problem D a function $F^D : \mathbb{B}^* \longrightarrow \mathbb{B}$ deciding D uniformly. Every decision problem D on $\mathbb{B}^*$ has a *non-uniform* variant consisting of the sequence $(F_k^D)_{k\in\mathbb{N}}$ of restrictions of F^D to $\mathbb{B}^k$ (see also [14]). In this section, we study the complexity of computing decision problems non-uniformly.

In the sequel, we denote by $\mathcal{IS}^{lf}(A)$ the set of *loop-free* $\mathrm{PGLB}_{\mathrm{bt}}(A)$ instruction sequences, i.e. the set of $\mathrm{PGLB}_{\mathrm{bt}}(A)$ instruction sequences without backward jumps. Moreover, we write *length*(I) for the number of primitive instructions of $I \in \mathcal{IS}(A)$.

Definition 3.1 Let $F : \mathbb{B}^k \longrightarrow \mathbb{B}$ be a k-ary function on the Booleans $\mathbb{B}$. $I \in \mathcal{IS}(A)$ is said to *compute F using l auxiliary registers* if for all $b_1, \ldots, b_k \in \mathbb{B}$

$$\left(|I|/\bigoplus_{i=1}^{l} \mathtt{aux}{:}i.B(\mathtt{t})\right)! \bigoplus_{i=1}^{k} \mathtt{in}{:}i.B(b_i) = F(b_1, \ldots, b_k).$$

Here we let $\{\mathtt{in}{:}i.B(b_i) \mid 1 \le i \le k\}$ and $\{\mathtt{aux}{:}i.B(\mathtt{t}) \mid 1 \le i \le l\}$ be two disjoint sets of Boolean registers and put $\bigoplus_{i=1}^{k} \mathtt{in}{:}i.B(b_i) = \{\mathtt{in}{:}i.B(b_i) \mid 1 \le i \le k\}$ and $\bigoplus_{i=1}^{l} \mathtt{aux}{:}i.B(\mathtt{t}) = \{\mathtt{aux}{:}i.B(\mathtt{t}) \mid 1 \le i \le l\}$. 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 $l = 0$.

Threads obtained from loop-free instruction sequences which reply to Boolean registers without the use of auxiliary registers are *binary decision diagrams* or *branching programs* [21]. They can represent every Boolean function.

Proposition 3.2 *Let $F : \mathbb{B}^k \longrightarrow \mathbb{B}$ be a k-ary 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 $\mathtt{t}$ or $\mathtt{f}$. Thus we can take for I_F either $!\mathtt{t}$ or $!\mathtt{f}$. 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 for $I_F = -\mathtt{in}{:}k+1.\mathtt{get}; \#3 \times 2^k - 1; I_{G_\mathtt{t}}; I_{G_\mathtt{f}}$ we have

$$(|I_F|/\emptyset)! \bigoplus_{i=1}^{k+1} \mathtt{in}{:}i.B(b_i) = \begin{cases} (|I_{G_\mathtt{t}}|/\emptyset)! \bigoplus_{i=1}^{k} \mathtt{in}{:}i.B(b_i) & \text{if } b_{k+1} = \mathtt{t}, \\ (|I_{G_\mathtt{f}}|/\emptyset)! \bigoplus_{i=1}^{k} \mathtt{in}{:}i.B(b_i) & \text{if } b_{k+1} = \mathtt{f}. \end{cases}$$

Thus I_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 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 decision problem.

Definition 3.3 For $F : \mathbb{B}^* \longrightarrow \mathbb{B}$, we denote by F_k ($k \in \mathbb{N}$) the restriction of F to $\mathbb{B}^k$ and distinguish the following classes of decision problems.

1. $\mathsf{IS}_P^{\mathit{lf},n} = \{F : \mathbb{B}^* \longrightarrow \mathbb{B} \mid$ there exists a polynomial function $h : \mathbb{N} \to \mathbb{N}$ such that for all $k \in \mathbb{N}$, F_k can be computed with n auxiliary registers by an $I \in \mathcal{IS}^{\mathit{lf}}(A)$ with $\mathit{length}(I) \leq h(k)\}$
2. $\mathsf{IS}_P^{\mathit{lf}} = \{F : \mathbb{B}^* \longrightarrow \mathbb{B} \mid$ there exists a polynomial function $h : \mathbb{N} \to \mathbb{N}$ such that for all $k \in \mathbb{N}$, F_k can be computed by an $I \in \mathcal{IS}^{\mathit{lf}}(A)$ with $\mathit{length}(I) \leq h(k)\}$
3. $\mathsf{IS}_P^{n} = \{F : \mathbb{B}^* \longrightarrow \mathbb{B} \mid$ there exists a polynomial function $h : \mathbb{N} \to \mathbb{N}$ such that for all $k \in \mathbb{N}$, F_k can be computed with n auxiliary registers by an $I \in \mathcal{IS}(A)$ with $\mathit{length}(I) \leq h(k)\}$
4. $\mathsf{IS}_P = \{F : \mathbb{B}^* \longrightarrow \mathbb{B} \mid$ there exists a polynomial function $h : \mathbb{N} \to \mathbb{N}$ such that for all $k \in \mathbb{N}$, F_k can be computed by an $I \in \mathcal{IS}(A)$ with $\mathit{length}(I) \leq h(k)\}$

Similarly, we define the classes $\mathsf{IS}_E^{\mathit{lf},n}$ ($n \in \mathbb{N}$), $\mathsf{IS}_E^{\mathit{lf}}$ and IS_E where $h : \mathbb{N} \to \mathbb{N}$ is now defined by $h(k) = c \times 2^k$ for some $c \in \mathbb{N}$.

From Proposition 3.2 it follows that this hierarchy of complexity classes collapses at the high end.

Proposition 3.4

$$\left.\begin{array}{r} \mathsf{IS}_P^{\mathit{lf},0} \subseteq \cdots \subseteq \mathsf{IS}_P^{\mathit{lf},n} \subseteq \cdots \subseteq \mathsf{IS}_P^{\mathit{lf}} \\ \mathsf{IS}_P^{0} \subseteq \cdots \subseteq \mathsf{IS}_P^{n} \subseteq \cdots \end{array}\right\} \subseteq \mathsf{IS}_P \subseteq \mathsf{IS}_E^{\mathit{lf},0} = \cdots = \mathsf{IS}_E^{\mathit{lf},n} = \cdots = \mathsf{IS}_E^{\mathit{lf}} = \mathsf{IS}_E.$$

$\mathsf{P/poly}$ is the complexity class of decision problems decided in polynomial time by non-uniform deterministic Turing machines with a polynomial-bounded advice function. It is also equivalently defined as the class PSIZE of problems that have polynomial-size Boolean circuits (see e.g. [15, 20]).

Theorem 3.5 $\mathsf{IS}_P^{\mathit{lf}} = \mathsf{P/poly}$

Proof This proof is an adaptation of the proofs of Theorems 5 and 6 given in [6] to our setting. 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 $F \in \mathsf{IS}_P^{lf}$. Then, for all $k \in \mathbb{N}$, there exists an $I_k \in \mathcal{IS}^{lf}(A)$ that computes F_k with *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 $\mathsf{P/poly}$.

$\supseteq$: 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 $\{g_1, \ldots, g_n\}$ ($n \in \mathbb{N}$) is a topological ordering of the gates with output node g_n, we prove by induction on n that we may assume that I_C is of the form $I; +\mathtt{aux}{:}n.\mathtt{get}; !\mathtt{t}; !\mathtt{f}$ 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

$$I_\neg = +\mathtt{in}{:}i.\mathtt{get}; \mathtt{aux}{:}1.\mathtt{set}{:}\mathtt{f}; +\mathtt{aux}{:}1.\mathtt{get}; !\mathtt{t}; !\mathtt{f},$$
$$I_\wedge = -\mathtt{in}{:}i.\mathtt{get}; \#2; -\mathtt{in}{:}j.\mathtt{get}; \mathtt{aux}{:}1.\mathtt{set}{:}\mathtt{f}; +\mathtt{aux}{:}1.\mathtt{get}; !\mathtt{t}; !\mathtt{f}, \text{ or}$$
$$I_\vee = +\mathtt{in}{:}i.\mathtt{get}; \#3; -\mathtt{in}{:}j.\mathtt{get}; \mathtt{aux}{:}1.\mathtt{set}{:}\mathtt{f}; +\mathtt{aux}{:}1.\mathtt{get}; !\mathtt{t}; !\mathtt{f}$$

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_n is an AND gate. Suppose that the input of g_n are the output gates g_l and g_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 $I_{C'} = I'; +\mathtt{aux}{:}l.\mathtt{get}; !\mathtt{t}; !\mathtt{f}$ and $I_{C''} = I''; +\mathtt{aux}{:}m.\mathtt{get}; !\mathtt{t}; !\mathtt{f}$ 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

$$I_C = I'; I''; -\mathtt{aux}{:}l.\mathtt{get}; \#2; -\mathtt{aux}{:}m.\mathtt{get}; \mathtt{aux}{:}n.\mathtt{set}{:}\mathtt{f}; +\mathtt{aux}{:}n.\mathtt{get}; !\mathtt{t}; !\mathtt{f}$$

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

Now suppose that $F \in \mathsf{P/poly}$. 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 IS_P^{lf}. □

In the remainder of this section we shall consider the separation of the remaining complexity classes. The following results were suggested by an anonymous reviewer of earlier versions of this paper.

Proposition 3.6 *The hierarchy also collapses at the low end, i.e. for all* $n, m \in \mathbb{N}$, $\mathsf{IS}_P^{lf,n} = \mathsf{IS}_P^m$.

Proof We shall first prove (1) $\mathsf{IS}_P^{m+1} \subseteq \mathsf{IS}_P^m$ and (2) $\mathsf{IS}_P^0 \subseteq \mathsf{IS}_P^{lf,0}$.

(1) Let $F \in \mathsf{IS}_P^{m+1}$. Without loss of generality we may assume that I computing F_k uses the auxiliary registers $B_1, \ldots, B_{m+1}$. Suppose $I = u_1; \cdots; u_n$ with $n \leq h(k)$. We then simulate I by $I' = \psi^{\mathtt{t}}(u_1); \cdots; \psi^{\mathtt{t}}(u_n); \psi^{\mathtt{f}}(u_1); \cdots; \psi^{\mathtt{f}}(u_n)$ where $\psi^b(u_1); \cdots; \psi^b(u_n)$ corresponds to an execution of I with register B_{m+1} holding the value b. This can be achieved by defining

$$\psi^{\mathtt{t}}(u) \equiv \begin{cases} \#1 & \text{if } u \in \{\mathtt{aux}{:}m+1.\mathtt{set{:}t}, +\mathtt{aux}{:}m+1.\mathtt{set{:}t}, \mathtt{aux}{:}m+1.\mathtt{get}, \\ & \qquad +\mathtt{aux}{:}m+1.\mathtt{get}\}, \\ \#2 & \text{if } u \in \{-\mathtt{aux}{:}m+1.\mathtt{set{:}t}, -\mathtt{aux}{:}m+1.\mathtt{get}\}, \\ \#n & \text{if } u \in \{\mathtt{aux}{:}m+1.\mathtt{set{:}f}, +\mathtt{aux}{:}m+1.\mathtt{set{:}f}, \\ & \qquad -\mathtt{aux}{:}m+1.\mathtt{set{:}f}\}, \text{ and} \\ u & \text{otherwise,} \end{cases}$$

$$\psi^{\mathtt{f}}(u) \equiv \begin{cases} \#1 & \text{if } u \in \{\mathtt{aux}{:}m+1.\mathtt{set{:}f}, \\ & \qquad -\mathtt{aux}{:}m+1.\mathtt{set{:}f}, \mathtt{aux}{:}m+1.\mathtt{get}, -\mathtt{aux}{:}m+1.\mathtt{get}\}, \\ \#2 & \text{if } u \in \{+\mathtt{aux}{:}m+1.\mathtt{set{:}f}, +\mathtt{aux}{:}m+1.\mathtt{get}\}, \\ \backslash\#n & \text{if } u \in \{\mathtt{aux}{:}m+1.\mathtt{set{:}t}, +\mathtt{aux}{:}m+1.\mathtt{set{:}t}, \\ & \qquad -\mathtt{aux}{:}m+1.\mathtt{set{:}t}\}, \text{ and} \\ u & \text{otherwise.} \end{cases}$$

Then I' computes F_k without the use of register B_{m+1} and $\mathit{length}(I') \leq 2 \times h(k)$. Hence $\mathsf{IS}_P^{m+1} \subseteq \mathsf{IS}_P^m$.

(2) Let $F \in \mathsf{IS}_P^0$. Without loss of generality we may assume that $\mathtt{aux}{:}i.m$, $+\mathtt{aux}{:}i.m$, $-\mathtt{aux}{:}i.m$ do not occur in I computing F_k. Observe that since input registers are read-only their Boolean reply is constant. It follows that since an execution of I terminates it will not visit an instruction twice. Thus every execution of I terminates in at most $\mathit{length}(I)$ steps. We can therefore replace backward jumps by forward jumps as follows. Suppose $I = u_1; \cdots; u_n$ with $n \leq h(k)$ and put

$$I' = \underbrace{\psi(u_1); \cdots; \psi(u_n); \cdots; \psi(u_1); \cdots; \psi(u_n)}_{n\times}$$

where

$$\psi(u) \equiv \begin{cases} \#n-l & \text{if } u \equiv \backslash\#l, \\ u & \text{otherwise.} \end{cases}$$

Then I' computes F_k without the use of backward jumps and $\mathit{length}(I') \leq (h(k))^2$. Hence $\mathsf{IS}_P^0 \subseteq \mathsf{IS}_P^{lf,0}$.

From (1) and (2) in now follows that $\mathsf{IS}_P^m \subseteq \mathsf{IS}_P^0 \subseteq \mathsf{IS}_P^{lf,0} \subseteq \mathsf{IS}_P^{lf,n}$. Also $\mathsf{IS}_P^{lf,n} \subseteq \mathsf{IS}_P^n \subseteq \mathsf{IS}_P^0 \subseteq \mathsf{IS}_P^m$. Thus $\mathsf{IS}_P^{lf,n} = \mathsf{IS}_P^m$. □

L/poly is the complexity class of decision problems computable in logarithmic space by non-uniform deterministic Turing machines with a polynomial-bounded advice function. L/poly can also be characterized in terms of polynomial size branching

programs (see e.g. [12, 18, 19]). Since threads obtained from loop-free instruction sequences which reply to Boolean read-only registers without the use of auxiliary registers are branching programs, we have

Corollary 3.7 $\mathsf{IS}_P^{lf,0} = \mathsf{IS}_P^{lf} \Leftrightarrow \mathsf{L/poly} = \mathsf{P/poly}$

$\mathsf{PSPACE/poly}$ is the complexity class of decision problems computable in polynomial space by non-uniform deterministic Turing machines with the help of an advice of polynomial length. In [1], $\mathsf{PSPACE/poly}$ has been alternatively characterized by polynomial-size quantified Boolean formulae.

Theorem 3.8 $\mathsf{IS}_P = \mathsf{PSPACE/poly}$

Proof The inclusion $\subseteq$ is proved similarly to Theorem 3.5 by simulation. The size of the description and the space used is still polynomial. The numbers of steps, however, may not be polynomial because of possible loops.

$\supseteq$: We first show that a function $F : \mathbb{B}^k \to \mathbb{B}$ that is induced by a quantified Boolean formula ϕ constructed from variables, the constants $\mathtt{t}$, $\mathtt{f}$, the connectives $\neg$, $\wedge$ and $\vee$, and the quantifiers $\exists$ and $\forall$ can be computed by an $I_\phi \in \mathcal{IS}(A)$ with bounded length. To these ends we first distinguish two kinds of variables: the variables $x_1, x_2, \ldots$ admitting quantification which will be fed by auxiliary registers and the free variables $y_1, y_2, \ldots$ for which values will be substituted from the input registers. We thus store the variable x_i in register aux:i and the variable y_i in register in:i; the result is computed in register aux:0. We then prove by induction on the length n of ϕ that we may assume that I_ϕ is of the form I; +aux:0.get; !t; !f for some $I \in \mathcal{IS}(A)$ with $\mathit{length}(I) \le 7 \times n$.

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

$I_\mathtt{t}$ = aux:0.set:t; +aux:0.get; !t; !f,
$I_\mathtt{f}$ = aux:0.set:f; +aux:0.get; !t; !f,
I_{x_i} = +aux:i.get; #3; aux:0.set:f; #2; aux:0.set:t; +aux:0.get; !t; !f, or
I_{y_i} = +in:i.get; #3; aux:0.set:f; #2; aux:0.set:t; +aux:0.get; !t; !f

comply. For the induction step we have to distinguish 5 cases. We here consider only the cases $\neg$, $\wedge$ and $\forall$.

If F is induced by $\neg\phi$, we may assume that the function induced by ϕ can be computed by the $\mathcal{IS}(A)$ instruction sequences $I_\phi = I'$; +aux:0.get; !t; !f with $\mathit{length}(I') \le 7 \times (n-1)$. Then

$I_{\neg\phi} = I'$; +aux:0.get; #3; aux:0.set:t; #2; aux:0.set:f; +aux:0.get; !t; !f

computes F and $\mathit{length}(I) = \mathit{length}(I') + 5 \le 7 \times n$.

Suppose that F is induced by $\phi_1 \wedge \phi_2$. By the induction hypothesis we may assume that the functions induced by ϕ_1 and ϕ_2 can be computed by the $\mathcal{IS}(A)$ instruction sequences $I_{\phi_1} = I_1$; +aux:0.get; !t; !f and $I_{\phi_2} = I_2$; +aux:0.get; !t; !f with $\mathit{length}(I_1) \le 7 \times n_1$ and $\mathit{length}(I_2) \le 7 \times n_2$ where n_1 and n_2 are the lengths of the

respective formulae. Then

$$I_\phi = I_1; -\texttt{aux:}0.\texttt{get}; \#(length(I_2)+1); I_2; +\texttt{aux:}0.\texttt{get}; !\texttt{t}; !\texttt{f}$$

computes F and $length(I) = length(I_1) + length(I_2) + 2 \leq 7 \times n_1 + 7 \times n_2 + 2 \leq 7 \times n$.

If F is induced by $\forall x_i \phi'$ we assume without restrictions to the general case that x_i does not occur quantified in ϕ'. Exploiting the fact that ϕ is logically equivalent with $\phi'[x_i := \texttt{t}] \wedge \phi'[x_i := \texttt{f}]$ we put

$$\begin{aligned} I_{\forall x_i \phi'} = {} & \texttt{aux:}i.\texttt{set:t}; I'; -\texttt{aux:}0.\texttt{get}; \#5; -\texttt{aux:}i.\texttt{get}; \#3; \texttt{aux:}i.\texttt{set:f}; \\ & \backslash\#(length(I')+5); +\texttt{aux:}0.\texttt{get}; !\texttt{t}; !\texttt{f} \end{aligned}$$

where $I_{\phi'} = I'; +\texttt{aux:}0.\texttt{get}; !\texttt{t}; !\texttt{f}$ and $length(I') \leq 7 \times (n-1)$. Observe that on entry x_i is set to $\texttt{t}$ and, if $\phi'[x_i := \texttt{t}]$ is $\texttt{f}$ execution terminates; otherwise x_i is set to $\texttt{f}$ and the result depends on $\phi'[x_i := \texttt{f}]$. Thus $I_{\forall x_i \phi'}$ computes F and $length(I) = length(I') + 7 \leq 7 \times n$.

Now suppose that $F \in \mathsf{PSPACE/poly}$. Then there exists a polynomial $p(k)$ such that for all $k \in \mathbb{N}$, there exists a quantified Boolean formula ϕ of length $p(k)$ which induces F_k. From the above and the fact that linear in the length of ϕ implies polynomial in k, it follows that F is also in IS_P. □

An immediate corollary is

Corollary 3.9 $\mathsf{IS}_P^{lf} = \mathsf{IS}_P \Leftrightarrow \mathsf{P/poly} = \mathsf{PSPACE/poly}$

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. 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 [13].

Corollary 3.10 $3SAT \in \mathsf{IS}_P$

Proof $3SAT \in \mathsf{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{PSPACE/poly}$. □

Finally, the remaining inclusion in Proposition 3.4 is proper.

Theorem 3.11 $\mathsf{IS}_P \subsetneq \mathsf{IS}_E$

Proof We employ a standard counting argument. There are 2^{2^k} Boolean functions $F : \mathbb{B}^k \to \mathbb{B}$. Under the assumption that in an instruction sequence of length n that computes a $k-ary$ Boolean function do neither occur instances of $\texttt{aux:}i.m$, $+\texttt{aux:}i.m$, $-\texttt{aux:}i.m$ for $i > n$ or $\texttt{in:}i.m$, $+\texttt{in:}i.m$, $-\texttt{in:}i.m$ for $i > k$ ($m \in \mathcal{M}$) or jumps $\#l$, $\backslash\#l$ for $l \geq n$, there are only $2^{n\,O(\log(n+k))}$ syntactically different instruction sequences of length n. So for a suitable natural number n with $\frac{2^{k-1}}{k} < n < \frac{2^k}{k}$ we cannot

compute all of the Boolean functions with instruction sequences with at most n instructions. Now if we let F_k be the first function in a lexicographically ordered list of all function tables that cannot be computed with n or fewer instructions, then we have a family $F = (F_k)_{k\in\mathbb{N}}$ of Boolean functions that cannot be polynomial bounded. □

4 Conclusion

One of the major goals of complexity theory is to separate complexity classes. We have investigated the possibility of separating classes with the aid of instruction sequences, and in particular, studied the contribution backward jumps can provide within the framework of PGA. There are several directions to continue the exploration of instruction sequences and services with respect to the question of expressiveness.

We only considered Boolean registers as services. We expect that more powerful services such as counters or stacks (cf. [3, 8]) define complexity classes that need more powerful Turing machines. Moreover, general instruction sequences describe programs that can become inactive. This can be exploited for a systematic investigation of the changes|if any|for instruction sequences and related Turing machines caused by partiality.

Acknowledgement The authors wish to thank the anonymous reviewers of earlier versions for thoroughly reading this paper and providing thoughtful recommendations.

Open Access This article is distributed under the terms of the Creative Commons Attribution Noncommercial License which permits any noncommercial use, distribution, and reproduction in any medium, provided the original author(s) and source are credited.

References

1. Balcazar, J.L., Diaz, J., Gabarro, J.: On characterizations of the class PSPACE/poly. Theor. Comput. Sci. **52**, 251–267 (1987)
2. Bergstra, J.A., Bethke, I.: Polarized process algebra and program equivalence. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30–July 4. LNCS, vol. 2719, pp. 1–21. Springer, Berlin (2003)
3. Bergstra, J.A., Bethke, I., Ponse, A.: Decision problems for pushdown threads. Acta Inform. **44**(2), 75–90 (2007)
4. Bergstra, J.A., Loots, M.E.: Program algebra for sequential code. J. Log. Algebr. Program. **51**(2), 125–156 (2002)
5. Bergstra, J.A., Middelburg, C.A.: Thread algebra for strategic interleaving. Form. Asp. Comput. **19**(4), 445–474 (2007)
6. Bergstra, J.A., Middelburg, C.A.: Instruction sequences and non-uniform complexity theory. arXiv:0809.0352v3 (2010)
7. Bergstra, J.A., Middelburg, C.A.: Instruction sequence processing operators. arXiv:0910.5564v3 (2010)
8. Bergstra, J.A., Ponse, A.: Combining programs and state machines. J. Log. Algebr. Program. **51**, 175–192 (2002)
9. Bui, D.B., Mavlyanov, A.V.: Theory of program algebras. Ukr. Math. J. **36**(6), 761–764 (1984)
10. Bui, D.B., Mavlyanov, A.V.: Mutual derivability of operations in program algebra. I. Cybern. Syst. Anal. **24**(1), 35–39 (1988)

11. Bui, D.B., Mavlyanov, A.V.: Mutual derivability of operations in program algebra. II. Cybern. Syst. Anal. **24**(6), 1–6 (1988)
12. Cobham, A.: The recognition problem for the set of perfect squares. Research paper RC-1704, IBM Watson Research Centre (1966)
13. Cook, S.: The complexity of theorem-proving procedures. In: Proc. 3rd STOC, pp. 151–158. IEEE Comput. Soc., Los Alamitos (1971)
14. Karp, R.M., Lipton, R.J.: Some connections between nonuniform and uniform complexity classes. In: Proc. 12th STOC, pp. 302–309. ACM, New York (1980)
15. Lutz, J.H., Schmidt, W.J.: Circuit size relative to pseudorandom oracles. Theor. Comput. Sci. **107**(1), 95–120 (1993)
16. Mealy, G.H.: A method for synthesizing sequential circuits. Bell Syst. Tech. J. **34**, 1045–1079 (1955)
17. Ponse, A., van der Zwaag, M.B.: An introduction to program and thread algebra. In: Beckmann, A., et al. (ed.) Logical Approaches to Computational Barriers: Proceedings CiE 2006. LNCS, vol. 3988, pp. 445–458. Springer, Berlin (2006)
18. Pudlak, P., Zak, S.: Space complexity of computations. Preprint, University of Prague (1983)
19. Savitch, W.: Relations between nondeterministic and deterministic tape complexities. J. Comput. Syst. Sci. **4**, 177–192 (1970)
20. Schöning, U.: Complexity and Structure. Springer, Berlin (1986)
21. Wegener, I.: Branching Programs and Binary Decision Diagrams. SIAM Monographs on Discrete Mathematics and Applications (2000)
22. von Wright, J.: An Interactive Metatool for Exploring Program Algebras. Turku Centre for Computer Science, TUCS Technical Report No. 247, March (1999)