Decision Problems for Pushdown Threads

J.A. Bergstra
I. Bethke
A. Ponse
J.A. Bergstra
Programming Research Group
Faculty of Science
University of Amsterdam
Kruislaan 403
1098 SJ Amsterdam
The Netherlands
tel. +31 20 525.7591
e-mail: janb@science.uva.nl

I. Bethke
Programming Research Group
Faculty of Science
University of Amsterdam
Kruislaan 403
1098 SJ Amsterdam
The Netherlands
tel. +31 20 525.7583
e-mail: inge@science.uva.nl

A. Ponse
Programming Research Group
Faculty of Science
University of Amsterdam
Kruislaan 403
1098 SJ Amsterdam
The Netherlands
tel. +31 20 525.7592
e-mail: alban@science.uva.nl

Programming Research Group Electronic Report Series
Decision Problems for Pushdown Threads

Jan A. Bergstra1,2, Inge Bethke1, and Alban Ponse1

1 University of Amsterdam, Faculty of Science, Programming Research Group*
2 Utrecht University, Department of Philosophy, Applied Logic Group

Abstract. Threads as contained in a thread algebra emerge from the behavioral abstraction from programs in an appropriate program algebra. Threads may make use of services such as stacks, and a thread using a single stack is called a pushdown thread. Equivalence of pushdown threads is shown decidable whereas pushdown thread inclusion is undecidable. This is again an example of a borderline crossing where the equivalence problem is decidable, whereas the inclusion problem is not.

1 Introduction

A challenging question in language theory is to decide whether the languages accepted by two different machines in some given class are the same. This question is called the equivalence problem. Another important question, known as the inclusion problem, is that of determining whether one language is a subset of another. The most obvious connection between these two problems is that the latter implies the former, that is, that any algorithm that decides inclusion for some family of languages can also be used to decide equivalence. The question then arises whether the converse holds, i.e., whether there are natural examples of language families with a decidable equivalence problem and an undecidable inclusion problem.

In 1973, Bird [8] found that the languages accepted by two-tape Rabin and Scott machines possess a decidable equivalence problem and an undecidable inclusion problem. Valiant explored this question further, finding two other families exhibiting this feature: the languages accepted by deterministic finite-turn pushdown automata [23] and deterministic one-counter pushdown automata [24]. In 1976, Friedman [10] investigated another subclass of deterministic pushdown automata—simple machines that have only one state and operate in real-time—and showed that these languages indeed have an undecidable inclusion problem. More recently, e.g., erasing and nonerasing pattern languages [14, 18] and deterministic context-free languages [9] have been added to this growing list.

In this paper we investigate yet another class: pushdown threads, a form of processes describing sequential program behaviour and using the services offered by a single stack. In this approach, threads as contained in a thread algebra emerge from the behavioral abstraction of sequential programs. A basic thread

* www.science.uva.nl/~janb, inge, alban. We thank the organizers of the Logic Colloquium 2005 in Athens for providing the opportunity to present this paper.
models a finite program behaviour to be controled by some execution environment: upon each action (e.g. a request for some service), a reply true or false from the environment determines further execution. Any execution trace of a basic thread ends either in the (successful) termination state or in the deadlock state. Both these states are modeled as special thread constants. Regular threads extend basic threads by comprising loop behaviour, and are reminiscent of flowcharts [15,11]. Threads may make use of services, i.e., devices that control (part of) their execution by consuming actions, providing the appropriate reply, and suppressing observable activity. Regular threads using the service of a single stack are called pushdown threads. Apart from the distinction between deadlock and termination, pushdown threads are comparable to pushdown automata. We show that equivalence of pushdown threads is decidable, whereas pushdown thread inclusion is undecidable.

The paper is structured as follows: in Section 2, we outline the fundamental properties of thread algebra. In Section 3, we show that equivalence between pushdown threads is decidable by reducing the equivalence problem for deterministic pushdown automata [19,21,22] to our equivalence problem. In Section 4, we prove that inclusion is undecidable for pushdown threads. Here we reduce the halting problem for Minsky machines to the inclusion problem—an approach also taken in Jancar et al. [13]. In Section 5 we shortly discuss a programming notation for the specification of pushdown threads. The paper ends with some conclusions in Section 6.

2 Threads and services

Basic thread algebra [6], BTA, is a form of process algebra which is tailored for the description of sequential program behaviour. Based on a finite set of actions $A$, it has the following constants and operators:

- the termination constant $S$,
- the deadlock or inaction constant $D$,
- for each $a \in A$, a binary postconditional composition operator $- \triangleright a \triangleright -$.

We use action prefixing $a \circ P$ as an abbreviation for $P \triangleright a \triangleright P$ and take $\circ$ to bind strongest. Furthermore, for $n \in \mathbb{N}$ we define $a^n \circ P$ by $a^0 \circ P = P$ and $a^n+1 \circ P = a \circ (a^n \circ P)$.

The operational intuition behind this algebraic framework is that each action represents a command which is to be processed by the execution environment of the thread. More specifically, an action is taken as a command for a service offered by the environment. The processing of a command may involve a change of state of this environment. At completion of the processing of the command, the service concerned produces a reply value. This reply is either true or false and is returned to the thread under execution. The thread $P \triangleright a \triangleright Q$ will then proceed

---

*a In [5], basic thread algebra is introduced under the name basic polarized process algebra.
as $P$ if the processing of $a$ leads to the reply `true` indicating the successful processing of $a$, and it will proceed as $Q$ if the processing of $a$ leads to the unsuccessful reply `false`.

BTA can be equipped with a partial order and an approximation operator in the following way:

1. $\sqsubseteq$ is the partial ordering on BTA generated by the clauses
   (a) for all $P \in \text{BTA}$, $D \sqsubseteq P$, and
   (b) for all $P_1, P_2, Q_1, Q_2 \in \text{BTA}$, $a \in A$,
   \[
P_1 \sqsubseteq Q_1 \land P_2 \sqsubseteq Q_2 \Rightarrow P_1 \sqsubseteq a \triangleright P_2 \sqsubseteq Q_1 \sqsubseteq a \triangleright Q_2.
   \]

2. $\pi : \mathbb{N} \times \text{BTA} \to \text{BTA}$ is the approximation operator determined by the equations
   (a) for all $P \in \text{BTA}$, $\pi(0, P) = D$,
   (b) for all $n \in \mathbb{N}$, $\pi(n + 1, S) = S$, $\pi(n + 1, D) = D$, and
   (c) for all $P, Q \in \text{BTA}, n \in \mathbb{N}$,
   \[
   \pi(n + 1, P \sqsubseteq a \triangleright Q) = \pi(n, P) \sqsubseteq a \triangleright \pi(n, Q).
   \]

We further write $\pi_n(P)$ instead of $\pi(n, P)$.

The operator $\pi$ finitely approximates every thread in BTA. That is, for all $P \in \text{BTA}$,
\[
\exists n \in \mathbb{N} \; \pi_0(P) \sqsubseteq \pi_1(P) \sqsubseteq \cdots \sqsubseteq \pi_n(P) = \pi_{n+1}(P) = \cdots = P.
\]

Every thread in BTA is finite in the sense that there is a finite upper bound to the number of consecutive actions it can perform. Following the metric theory of [1] in the form developed as the basis of the introduction of processes in [4], BTA has a completion $\text{BTA}^\infty$ which comprises also the infinite threads. Standard properties of the completion technique yield that we may take $\text{BTA}^\infty$ as the cpo consisting of all so-called projective sequences. That is,
\[
\text{BTA}^\infty = \{(P_n)_{n \in \mathbb{N}} \mid \forall n \in \mathbb{N} \; (P_n \in \text{BTA} \land \pi_n(P_{n+1}) = P_n)\}
\]
with
\[
(P_n)_{n \in \mathbb{N}} \sqsubseteq (Q_n)_{n \in \mathbb{N}} \iff \forall n \in \mathbb{N} \; P_n \sqsubseteq Q_n
\]
and
\[
(P_n)_{n \in \mathbb{N}} = (Q_n)_{n \in \mathbb{N}} \iff \forall n \in \mathbb{N} \; P_n = Q_n.
\]
For a detailed account of this construction see [2]. In this cpo structure, finite linear recursive specifications represent continuous operators having as unique fixed points regular threads, i.e., threads which can only reach finitely many states. A finite linear recursive specification over $\text{BTA}^\infty$ is a set of equations
\[
X_i = t_i(X)
\]
for $i \in I$ with $I$ some finite index set and all $t_i(X)$ of the form $S$, $D$, or $X_i \sqsubseteq a_i \triangleright X_i$, for $i, i_r \in I$.  

3
Example 1. We define the regular threads

1. \(a^n \circ D\),
2. \(a^n \circ S\) and
3. \(a^\infty\) (this informal notation will be often used in the sequel)

as the fixed points for \(X_1\) in the specifications

1. \(X_1 = a \circ X_2, \ldots, X_{n-1} = a \circ X_n, X_n = D\),
2. \(X_1 = a \circ X_2, \ldots, X_{n-1} = a \circ X_n, X_n = S\),
3. \(X_1 = a \circ X_1\), respectively.

Both \(a^n \circ D\) and \(a^n \circ S\) are finite threads; \(a^\infty\) is the infinite thread corresponding to the projective sequence \((P_n)_{n\in\mathbb{N}}\) with \(P_0 = D\) and \(P_{n+1} = a \circ P_n\). Observe that e.g. \(a^n \circ D \subseteq a^n \circ S\), \(a^n \circ D \subseteq a^\infty\) but \(a^n \circ S \not\subseteq a^\infty\).

In reasoning with finite linear specifications, we shall from now on identify variables and their fixed points. For example, we say that \(P\) is the regular thread defined by \(P = a \circ P\) instead of stating that \(P\) equals the fixed point for \(X\) in the finite linear specification \(X = a \circ X\). Furthermore, in a finite linear specification

\[
P_i = t_1(P), \ldots, P_n = t_n(P),
\]

the threads \(P_i\) will also be referred to as states. As another example following this convention, we show that for regular threads \(P\) and \(Q\), \(P \subseteq Q\) is decidable (recall that \(\subseteq\) is decidable for finite threads). Because one can always take the disjoint union of two recursive specifications, it suffices to show that \(P_i \not\subseteq P_j\) is decidable in (1) above. This follows with finite approximations from the following assertion (the idea being that any loop has at most length \(n\)):

\[
\forall i, j \leq n \; \pi_n(P_i) \subseteq \pi_n(P_j) \Rightarrow P_i \subseteq P_j
\]

(2)

where \(\pi_i(P_k)\) is defined by \(\pi_i(t_k(P))\). To prove (2), assume that \(n > 1\) (otherwise the implication follows trivially). Choose \(i, j\) and assume that \(\pi_n(P_i) \subseteq \pi_n(P_j)\). Suppose \(P_i \not\subseteq P_j\). Then for some \(k > n\), \(\pi_k(P_i) \not\subseteq \pi_k(P_j)\) while \(\pi_{k-1}(P_i) \subseteq \pi_{k-1}(P_j)\). So there exists a trace of length \(k\) from \(P_i\) of the form

\[
P_i \xrightarrow{a_n a_{n-1} \ldots a_1} P' \xrightarrow{b_{n-1} \ldots b_1} \ldots
\]

that is not a trace of \(P_j\), while by the assumption the first \(n\) actions of this trace are a trace of \(P_j\). These \(n\) actions are connected by \(n+1\) states, and because there are only \(n\) different states, a repetition occurs in this sequence of states. So the trace witnessing \(\pi_k(P_i) \not\subseteq \pi_k(P_j)\) can be made shorter, contradicting \(k\)'s minimality and hence the supposition. Thus \(P_i \not\subseteq P_j\). As a corollary, also \(P = Q\) is decidable for regular threads \(P\) and \(Q\).
A service is a pair \( \langle \Sigma, F \rangle \) consisting of a set \( \Sigma \) of special actions and a reply function \( F \). The reply function \( F \) of a service \( \langle \Sigma, F \rangle \) is a mapping that gives for each finite sequence of actions from \( \Sigma \) the reply produced by the service. This reply is a Boolean value true or false.

**Example 2.** Services that will occur in Section 3 and 4 are

1. \( C = \langle \Sigma, F \rangle \) with \( \Sigma = \{ \text{inc}, \text{dec} \} \) consisting of the increase and decrease actions of a natural number counter and the reply function \( F \) which always replies true to increase actions and false to decrease actions if and only if the counter is zero. We denote by \( C(n) \) a counter with value \( n \).

2. \( S = \langle \Sigma, F \rangle \) with \( \Sigma = \{ \text{push} : i, \text{topeq} : i, \text{empty, pop} \mid i = 1, \ldots, n \} \) for some \( n \) where \text{push} : i pushes \( i \) onto the stack and yields reply true, the action \text{topeq} : i tests whether \( i \) is on top of the stack, \text{empty} tests whether the stack is empty, and \text{pop} pops the stack if it is non-empty with reply true and yields false otherwise. We denote by \( S(\alpha) \) a stack with contents \( \alpha \in \{1, \ldots, n\}^* \).

Observe that counters can be seen as particular stacks (take \( n = 1 \)).

In order to provide a specific description of the interaction between a thread and a service, we will use for actions the general notation \( c.a \) where \( c \) is the so-called channel and \( a \) is the so-called co-action. In particular, we will write e.g. \( c.\text{inc} \) for the action which increases a counter via channel \( c \) and \( s.\text{pop} \) to denote the action which pops a stack via channel \( s \). For a service \( S = \langle \Sigma, F \rangle \) and a finite thread \( P \), the defining rules for \( P/c \ S \) (the thread \( P \) using the service \( S \) via channel \( c \) are:

\[
S/c \ S = S, \\
D/c \ S = D, \\
(P \leq c'.a \triangleright Q) /c \ S = (P/c \ S) \leq c'a \triangleright (Q/c \ S) \text{ if } c' \neq c, \\
(P \leq c.a \triangleright Q) /c \ S = P/c \ S' \text{ if } a \in \Sigma \text{ and } F(a) = \text{true}, \\
(P \leq c.a \triangleright Q) /c \ S = Q/c \ S' \text{ if } a \in \Sigma \text{ and } F(a) = \text{false}, \\
(P \leq c.a \triangleright Q) /c \ S = D \text{ if } a \not\in \Sigma.
\]

where \( S' = \langle \Sigma, F' \rangle \) with \( F'(\sigma) = F(\sigma a) \) for all co-action sequences \( \sigma \in \Sigma^* \). The use operator is expanded to infinite threads \( P \) by stipulating

\[
P/c \ S = (\pi_n(P)/c \ S)_{n \in \mathbb{N}}.
\]

As a consequence, \( P/c \ S = D \) if for any \( n, \pi_n(P)/c \ S = D \). Finally, repeated applications of the use operator bind to the left, thus \( P/c_0 S_0 /c_1 S_1 = (P/c_0 S_0) /c_1 S_1 \).

**Example 3.** We consider again the threads \( a^n \circ D, a^n \circ S \) and \( a^\infty \) from Example 1 but now in the versions \( c.a^n \circ D \) (short for \( (c.a)^n \circ D \)), \( c.a^n \circ S \) and \( c.a^\infty \) for some channel \( c \) and some service \( S = \langle \Sigma, F \rangle \) with \( a \in \Sigma \). Then \( (c.a^n \circ D) /c \ S = D, (c.a^n \circ S) /c \ S = S \) but \( c.a^\infty /c \ S = D \). The last identity can also be retrieved as

\[4\) In [7] a service is called a state machine.

5
follows: if $P = \sigma \circ P$, then $P/S = (\sigma \circ P)/S = P/S'$. In this way, the computation will diverge, characterizing the rule: “If for a regular thread $P$ the defining rules for $P/S$ fail to prove $P/S = S$ or $\pi(P/S) = b \circ D$ for some action $b$, then $P/S = D$.”

In the next example we show that the use of services may turn regular threads into irregular ones.

**Example 4.** Let $a \in A$. Consider the following regular process $P$: 5

$$P = (c_0 \cdot \text{inc} \circ P) \trianglelefteq a \triangleright ((P \trianglelefteq c_0 \cdot \text{dec} \triangleright D) \trianglelefteq a \triangleright (D \trianglelefteq c_0 \cdot \text{dec} \triangleright P)).$$

With the counter $C$ defined in Example 2, define for $n \in \mathbb{N}$ the thread $P_n$ by

$$P_n = P/c_0 C(n).$$

This definition yields the following infinite recursive specification:

$$\begin{align*}
P_0 &= P_1 \trianglelefteq a \triangleright (D \trianglelefteq a \triangleright P_0) \\
P_{k+1} &= P_{k+2} \trianglelefteq a \triangleright (P_k \trianglelefteq a \triangleright D) \text{ for } k \in \mathbb{N}.
\end{align*}$$

Now assume that $P_n \subseteq P_m$ for some $n > m$. Then, by definition of $P_m$ (following the rightmost branches of $\trianglelefteq a \triangleright$),

$$(P_{n-m-1} \trianglelefteq a \triangleright D) \not\subseteq (D \trianglelefteq a \triangleright P_0).$$

It is clear that $\subseteq$ does not hold, thus we obtained a contradiction and $P_n \not\subseteq P_m$. In the same way it follows that $P_n \not\supseteq P_m$ if $n > m$. We conclude that $P_0$ is not regular: for each $k \in \mathbb{N}$, the state $P_k$ can be reached, and if $n \neq m$, then the states $P_n$ and $P_m$ are different. 6

We call a regular thread that uses a stack or a counter as described in Example 2 a pushdown thread (typically, any thread $P_k$ defined in Example 4 is a pushdown thread).

### 3 Decidable equality

In this section we prove the decidability of $P/S(\alpha) = Q/S(\beta)$ for regular threads $P$ and $Q$, and a stack $S$ over some finite data type with contents $\alpha$, respectively $\beta$. We first discuss deterministic pushdown automata (dpda’s) and recall the decidability result establishing dpda-equivalence (Sénizergues [19,21]). We base our approach on Stirling’s account of this result in [22], as this is closer to our setting. In the sequel we write $\epsilon$ for the empty sequence over any alphabet.

---

5 Note that a linear recursive specification requires (at least) six equations in this case.

6 Note that for all $n$, $P_n \subseteq a^\infty$ (this follows from $\pi_k(P_n) \subseteq \pi_k(a^\infty) = a^k \circ D$ for all $k$) and $P_n \not\subseteq S$. We will use these properties in Section 4.
A pushdown automaton (pda) \( A \) is given by a finite set \( \mathcal{P} \) of states, a finite set \( \mathcal{S} \) of stack symbols, a finite alphabet \( \mathcal{A} \), and a finite set of basic transitions of the form 
\[
P x \xrightarrow{\alpha} Q \alpha
\]
with \( P, Q \in \mathcal{P}, x \in \mathcal{S}, \alpha \in \mathcal{A} \cup \{ \} \), and \( \alpha \in \mathcal{S}^* \). Expressions of the form \( P x \) are further called source-configurations. A configuration of \( A \) is any expression \( P \alpha \) whose behaviour is determined by the basic transitions and the prefix rule
\[
\text{if } P x \xrightarrow{\alpha} Q \alpha \text{ then } P x \beta \xrightarrow{\alpha} Q \alpha \beta.
\]
The language accepted by a configuration \( P \alpha \), notation
\[
L(A, P \alpha),
\]
is \( \{ w \in \mathcal{A}^* \mid \exists Q \in \mathcal{P} . P \alpha \xrightarrow{w} Q \} \) where the extended transitions for words are defined as expected. Note that \( \varepsilon \)-transitions are swallowed in the usual fashion and that acceptance is by empty stack.

A deterministic pushdown automaton (dpda) \( A' \) has three restrictions on its basic transitions:

- if \( P x \xrightarrow{\alpha} Q \alpha \) and \( P x \xrightarrow{\beta} R \beta \) for \( \alpha, \beta \in \mathcal{A} \), then \( Q = R \) and \( \alpha = \beta \),
- if \( P x \xrightarrow{\alpha} Q \alpha \) and \( P x \xrightarrow{\varepsilon} R \varepsilon \) then \( \alpha = \varepsilon \),
- if \( P x \xrightarrow{\alpha} Q \alpha \) then \( \alpha = \varepsilon \) (so \( \varepsilon \)-transitions can only pop the stack; in [21] this case is referred to as a normalized dpda).

Note that the language accepted by any configuration of a dpda \( A' \) is prefix-free: if \( w \) is accepted then no proper prefix of \( w \) is accepted. For dpda's it is decidable whether \( L(A', P \alpha) = L(A', Q \beta) \) ([19, 21]). With this decidability result we can prove the main result of this section:

**Theorem 1.** For regular threads \( P \) and \( Q \), and a stack \( S \) over a finite data type it is decidable whether
\[
P_S S(\alpha) = Q_S S(\beta),
\]
where \( \alpha, \beta \) represent the contents of \( S \).

**Proof.** Let \( S \) be the (empty) stack controlled by the actions
\[
\{ \text{s.push}: i, \text{s.top} \text{eq}: i, \text{s.empty}, \text{s.pop} \mid i = 1, \ldots, n \}
\]
for some \( n \geq 1 \). We write \( S(\alpha) \) if the stack contains the elements from sequence \( \alpha \in \{ 1, \ldots, n \}^* \) with the leftmost element of \( \alpha \) on top. Furthermore, \( \text{s.push}: i \) pushes \( i \) onto the stack and yields reply \text{true}, the action \( \text{s.top \text{eq}}: i \) tests whether \( i \) is on top of the stack, \( \text{s.empty} \) tests whether the stack is empty, and \( \text{s.pop} \) pops the stack if it is non-empty (reply \text{true} and yields \text{false} otherwise. Finally, we assume that in \( P \) and \( Q \) there are external actions \( a_0, \ldots, a_m \), and that \( P \) and \( Q \) are given by a single finite linear specification. Using five transformations, we reduce the dpda-equivalence problem as discussed above to the question
\[
P_S S(\alpha) = Q_S S(\beta).
\]
Adapting the stack contents. In order to use the dpda-equivalence result, the stack should be empty upon termination and non-empty upon the start because language acceptance is by empty stack, and defined on configurations with non-empty stack. This can be achieved as follows:

- Extend the stack $S$ to $S_0$ as follows:
  - in order to start with a non-empty stack, add an extra stack symbol $0$ and extend all reply functions to actions over stack symbols $\{0, 1, \ldots, n\}$ in the obvious way, except for $\text{s.pop}$ that leaves $0$ on the stack and yields in this case reply $\text{false}$,
  - define a new action $\text{s.pop:all}$ that pops any symbol in $\{0, 1, \ldots, n\}$ from the stack with reply $\text{true}$, and yields reply $\text{false}$ if the stack is empty.

- In the specification of $P$ and $Q$,
  - replace each occurrence of $\text{s.empty}$ by $\text{s.top} = 0$ (so, $0$ acts as an empty stack marker),
  - replace any equation $R = S$ by $R = \text{s.pop:all} \circ S$ (where $S$ is fresh),
  - add the equation $S = S \subseteq \text{s.pop:all} \supseteq S$.

Call the resulting threads $P_0$ and $Q_0$, respectively, and assume these are given by an appropriate adaptation of the linear specification of $P$ and $Q$ (note that $\text{s.push:0}$ and $\text{s.empty}$ do not occur in this specification). It follows straightforwardly that for $\alpha, \beta \in \{1, \ldots, n\}^*$,

$$P_0/s \ S(\alpha) = Q_0/s \ S(\beta) \iff P_0/s \ S_0(\alpha 0) = Q_0/s \ S_0(\beta 0)$$

and that upon $S$ (i.e., termination), the stack $S_0$ is empty.

Replacement of $D$ by explicit loops. In the specification of $P_0$ and $Q_0$, replace each equation $R = D$ by $R = \text{s.push:1} \circ L$ (where $L$ is fresh) and add the equation $L = \text{s.push:1} \circ L$. Call the resulting threads $P_1$ and $Q_1$, respectively, for some appropriate adaptation to a linear recursive specification. Again it is straightforward that for $\alpha, \beta \in \{1, \ldots, n\}^*$,

$$P_0/s \ S_0(\alpha 0) = Q_0/s \ S_0(\beta 0) \iff P_1/s \ S_0(\alpha 0) = Q_1/s \ S_0(\beta 0)$$

and that upon $S$, the stack $S_0$ is empty.

Normalization of infinite traces. Let $\text{halt}$ be a fresh external action. Replace in $P_1$ and $Q_1$’s specification each equation $R = R_i \preceq a \succeq R_r$ with $a$ an external action by $R = S_0 \preceq \text{halt} \supseteq (R_i \preceq a \preceq R_r)$. Call the resulting threads $P_2$ and $Q_2$, respectively. Again it is straightforward that for $\alpha, \beta \in \{1, \ldots, n\}^*$,

$$P_1/s \ S_0(\alpha 0) = Q_1/s \ S_0(\beta 0) \iff P_2/s \ S_0(\alpha 0) = Q_2/s \ S_0(\beta 0)$$

and that upon $S$, the stack $S_0$ is empty. Moreover, each infinite sequence of external actions in $P_1/s \ S_0(\alpha 0)$ or $Q_1/s \ S_0(\beta 0)$ becomes after this transformation interlarded with $\text{halt} \circ S$ exits, so gives rise to an infinite number of (finite) traces.
Transformation to pd-a-equivalence. From the linearized specification of \( P_2 \) and \( Q_2 \), construct a pd-a \( A_1 \) as follows: for \( P \), the set of states, take those of the linear specification; for \( S \), the set of stack symbols, take \( \{0,\ldots,n\} \); and for the alphabet \( \Sigma \) take \( \{a_{\text{true}}, a_{\text{false}} | \ a \text{ an external action in } P_2 \text{ or } Q_2 \} \). As for the basic transitions,

- for each state \( R = R_i \leq a \geq R_r \) with \( a \) an external action and \( i \in \{1,\ldots,n\} \),
define transitions \( R_i \xrightarrow{a_{\text{true}}} R_i \) and \( R_i \xrightarrow{a_{\text{false}}} R_r \),
- for each state \( R = R_i \leq s \cdot \text{push} \cdot j \geq R_r \) and \( i \in \{1,\ldots,n\} \), define transitions \( R_i \xrightarrow{s} R_j \) with \( j \in \{1,\ldots,n\} \),
- for each state \( R = R_i \leq s \cdot \text{top} \cdot eq \cdot j \geq R_r \) and \( i \in \{0,1,\ldots,n\} \), define transitions \( R_i \xrightarrow{j} R_i \) and \( R_i \xrightarrow{\text{pop}} R_r \),
- for each state \( R = R_i \leq s \cdot \text{pop} \cdot \text{all} \geq R_r \) and \( i \in \{0,1,\ldots,n\} \), define transitions \( R_i \xrightarrow{\text{pop}} R_i \).

It follows immediately that for \( \alpha, \beta \in \{1,\ldots,n\}^* \),

\[
P_{2/\alpha} S_0(\alpha 0) = Q_{2/\alpha} S_0(\beta 0) \Leftrightarrow L(A_1, P_{2/\alpha} 0) = L(A_1, Q_{2/\alpha} 0).
\]

Note that \( A_1 \)'s basic transitions with label \( \epsilon \) are very limited: either a push or a pop, or they do not change the stack. Also, the transition relation is deterministic: for each source configuration \( R_i \) at most one transition is defined.

Transformation to dpda-equivalence. The only remaining problem is that \( A_1 \) is not normalized, i.e., may have \( \epsilon \)-transitions that do not pop the stack. However, the set of basic \( \epsilon \)-transitions can be transformed to one that contains only popping \( \epsilon \)-transitions while preserving language acceptance. Not swallowing \( \epsilon \)-transitions, each source-configuration \( R_i \) in \( A_1 \) with an \( \epsilon \)-transition can be classified by tracing its consequent transitions. Assume \( A_1 \) has \( k \) states. Because \( A_1 \) has \( n+1 \) stack symbols, it is sufficient to consider at most \( k(n+1)+1 \) consecutive transitions. Let \( \xrightarrow{\epsilon} \) stand for at least 0 and at most \( k(n+1)-1 \) \( \epsilon \)-transitions, then the following classes can be distinguished:

1. \( R_i \xrightarrow{\epsilon} R' \xrightarrow{\epsilon} R'' \xrightarrow{\epsilon} R''' \) where \( R'' \xrightarrow{\epsilon} R''' \) is the first occurring \( \lambda \)-transition,
2. \( R_i \xrightarrow{\epsilon} R' \xrightarrow{\epsilon} R'' \xrightarrow{\epsilon} R''' \) where \( R'' \xrightarrow{\epsilon} R''' \) is the first \( \epsilon \)-transition from \( Ri \) that empties the stack,
3. Not 1 or 2, i.e., \( R_i \xrightarrow{\epsilon} R'j \xrightarrow{\epsilon} R''m \xrightarrow{\epsilon} R''' \) and in the sequence of associated source-configurations a repetition occurs (and no empty stack).

Now define a dpda \( A_2 \) with \( P \), \( S \) and \( \Sigma \) as in \( A_1 \), and with basic transitions as follows:

- in case 1, replace \( R_i \xrightarrow{\epsilon} R' \) by \( R_i \xrightarrow{\epsilon} R''' \),
- in case 2, replace \( R_i \xrightarrow{\epsilon} R' \) by \( R_i \xrightarrow{\epsilon} R''' \),
- in case 3, simply omit the \( \epsilon \)-transition from \( R_i \),
keep all basic transitions with a label in $\mathbb{A}$ from $A_1$.

It is clear that $L(A_1, R\alpha) = L(A_2, R\alpha)$. In $A_2$, language equivalence is decidable as $A_2$ satisfies the dpda-definition given above (which is taken from [22]; note that the requirement in [22] that $|\alpha| < 3$ in a basic transition $P\xrightarrow{\alpha} Q\alpha$ can be trivially fulfilled by introducing auxiliary stack symbols).

We note that in the proof above, the transformation step Normalization of infinite traces can be skipped if one considers bisimulation equivalence in the transition graph of a normalized dpda, which is decidable as well (Sénizergues [20]).

We conclude this section with a short comment. The restriction to a stack over a finite data type is not essential for the decidability of equality between pushdown threads: also for a stack $S_N$ over the natural numbers $\mathbb{N}$ and regular threads $P$ and $Q$ it holds that $P/\mathbb{S} S_N(\alpha) = Q/\mathbb{S} S_N(\beta)$ is decidable. This can be seen by representing natural numbers in some unary notation and using a second data element as a separator. For example,

\begin{align*}
011101101111
\end{align*}

represents the stack containing $2, 1, 3$ (so, the natural number $n$ is represented by $n+1$ pushes of 1). For any $i \in \mathbb{N}$, the actions $\texttt{s.push}$ and $\texttt{s.top eq}$ can be expressed in this notation, be it a bit cumbersome. The action $\texttt{s.pop}$ is easier to define, and $\texttt{s.empty}$ need not be redefined. Thus given regular threads $P$ and $Q$, there exists a transformation $\phi$ (depending on the stack-actions in $P$ and $Q$) such that

\begin{align*}
P/\mathbb{S} S_N(\alpha) = Q/\mathbb{S} S_N(\beta) \Leftrightarrow \phi(P)/\mathbb{S} S(\overline{\alpha}) = \phi(Q)/\mathbb{S} S(\overline{\beta})
\end{align*}

where $S$ is the stack over symbols $\{0, 1\}$ and $\overline{\alpha}$ transforms a sequence of natural numbers as indicated above.

Successor and predecessor for $S_N$ can be easily defined using the representation discussed here. This is not the case for any action controlling $S_N$. For instance, an action $\texttt{swap}$ that exchanges the two top-elements of $S_N$ is not definable because with this action the functionality of a Minsky machine is obtained (using the even stack positions to hold the values of the first counter, and the odd positions for those of the second counter). So with $\texttt{swap}$ equality is not decidable. Neither is equality of the two top-elements a definable action. Of course, both these actions are definable for stacks over a finite data type.

4 Undecidable inclusion

The Minsky machine is a universal model of computation first used in [16, 17]. It is a simple imperative program consisting of a sequence of instructions labelled by natural numbers from 1 to some $L$. It starts from the instruction labelled 1,
halts if stop is reached and operates with two natural number counters \(c_0\) and \(c_1\). The instruction set consists of three types of instructions:

1. \(1: c_i := c_i + 1; \text{goto } l'\)
2. \(1: \text{if } c_i = 0 \text{ then goto } l' \text{ else } c_i := c_i - 1; \text{goto } l''\)
3. \(1: \text{stop}\)

where \(i \in \{0, 1\}\) and \(l, l', l'' \in \{1, \ldots, L\}\). It should be clear that the execution process is deterministic and has no failure. Any such process is either finished by the execution of the stop instruction or lasts forever. As expected, the halting problem for Minsky machines is undecidable:

**Theorem 2.** ([16, 17]) It is undecidable whether a Minsky machine halts when both counter values are initially zero.

In our setting, a counter \(C\) is a service \((\Sigma, F)\) with increase and decrease actions \(\Sigma = \{\text{inc}, \text{dec}\}\) and a reply function \(F\) which always replies \text{true} to \text{inc} and \text{false} to \text{dec} if and only if the counter value is zero. A Minsky machine canonically defines the equations of a specification of a regular thread:

\[
1: c_i := c_i + 1; \text{goto } l' \quad \Rightarrow \quad M_l = c_i.\text{inc} \circ M_{l'}
\]

\[
1: \text{if } c_i = 0 \text{ then goto } l' \quad \Rightarrow \quad M_l = M_{l'} \sqsubset c_i.\text{dec} \triangleright M_{l''}
\]

\[
1: \text{stop} \quad \Rightarrow \quad M_l = S.
\]

We call a thread \(M_l\) as defined above a \textit{Minsky thread}, thus a regular thread over \(S\) and the counter actions \(c_0.\text{inc}, c_0.\text{dec}, c_1.\text{inc}, c_1.\text{dec}\) where the channels \(c_0\) and \(c_1\) refer to the two internal counters \(C_0\) and \(C_1\), respectively. \(^7\) The halting problem for Minsky machines can now be rephrased to

**Theorem 3.** It is undecidable whether for a Minsky thread \(M\) it holds that

\[
M/c_0 C_0(0)/c_1 C_1(0) = S.
\]

(Of course, if \(M/c_0 C_0(0)/c_1 C_1(0) \neq S\), it equals \(D\).) The undecidability proof that follows consists of a reduction from the above halting problem to the inclusion problem for the use of a single internal counter—and thus to the inclusion problem for the use of a single internal stack.

For technical purposes we introduce the \textit{norm} of a Minsky thread operating on counters \(C_0(n)\) and \(C_1(m)\) as the number of counter actions until termination occurs (possibly \(\infty\)). The norm is formally defined with help of a transformation \(\theta\) of finite linear recursive specifications.

**Definition 1 (norm).** Let \(M_l\) be a Minsky thread defined by a finite linear recursive specification \(E = \{M_n = t_n(M) \mid n = 1, \ldots, k\}\) (for some \(k > 0\)) and let \(a\) be some action. Define \(\theta(E) = \{\theta(M_n) = \theta(t_n(M)) \mid n = 1, \ldots, k\}\) and define \(\theta(t_n(M))\) by

\(^7\) We do not have to disambiguate these counters because of the different channel names; however, we do this to enhance readability.
1. \( \theta(S) = S \)

2. For \( i \in \{0, 1\} \) and \( b \in \{ \text{inc, dec} \}, \)
   \[
   \theta(M_i \trianglelefteq c_i.b \trianglerighteq M_r) = a \circ \theta(M_i) \trianglelefteq c_i.b \trianglerighteq a \circ \theta(M_r).
   \]

For \( n, m \in \mathbb{N} \) define the norm \( \|M_1, n, m\| \in \mathbb{N} \cup \{ \infty \} \) by \( \|M_1, n, m\| = 0 \) if \( \theta(M_1) = S \in \theta(E) \), and for \( k > 0 \),
   \[
   \|M_1, n, m\| = k \quad \text{if} \quad \begin{cases} 
   \pi_{k+1}(\theta(M_1)/c_0 C_0(n)/c_1 C_1(m)) = a^k \circ S \\
   \pi_k(\theta(M_1)/c_0 C_0(n)/c_1 C_1(m)) = a^{k-1} \circ D.
   \end{cases}
   \]

For \( n, m \in \mathbb{N} \), \( \|M_1, n, m\| = \infty \) if for no \( k \in \mathbb{N} \), \( \|M_1, n, m\| = k \).

Note that
   \[
   \|M, n, m\| \in \mathbb{N} \Leftrightarrow M /_{c_0} C_0(n) /_{c_1} C_1(m) = S.
   \]

So the question whether \( \|M, n, m\| \in \mathbb{N} \) is undecidable.

We now introduce a transformation \( \psi \) of Minsky threads which replaces specific runs with the second counter \( C_1(0) \) by regular threads where the \( C_1 \)-actions are simulated by a single external action \( a \) in such a way that termination behaviour is preserved. Moreover, \( C_0 \)-actions are preceded by the simulation of a \( c_1.\text{inc} \circ c_1.\text{dec} \)-prefix. The reason for this is that divergence on \( C_0 \)-actions (as in \( c_0.\text{inc}^\infty \)) then also yields an infinite sequence of \( a \)-actions, which enables us to use a smooth proof strategy.

**Definition 2 (simulation).** Let \( M_1 \) be a Minsky thread defined by a finite linear recursive specification \( E = \{ M_n = t_n(M) \mid n = 1,\ldots,k \} \) (for some \( k > 0 \)). Define \( \psi(E) = \{ \psi(M_n) = \psi(t_n(M)) \mid n = 1,\ldots,k \} \) and define \( \psi(t_n(M)) \) by

1. \( \psi(S) = S \)

2. \( \psi(M_i \trianglelefteq c_1.\text{inc} \trianglerighteq M_j) = \psi(M_i) \trianglelefteq a \trianglerighteq a^\infty \)

3. \( \psi(M_i \trianglelefteq c_1.\text{dec} \trianglerighteq M_j) = a^\infty \trianglelefteq a \trianglerighteq (\psi(M_i) \trianglelefteq c_0.b \trianglerighteq \psi(M_j)) \)

4. \( \psi(M_i \trianglelefteq c_0.b \trianglerighteq M_j) = \)

\[
[a^\infty \trianglelefteq a \trianglerighteq (\psi(M_i) \trianglelefteq c_0.b \trianglerighteq \psi(M_j))] \trianglelefteq a \trianglerighteq a^\infty
\]

for \( b \in \{ \text{inc, dec} \} \).

We note that \( a^\infty /_{c_0} C_0(n) = a^\infty \) and therefore we omit any such use-application in reasoning about \( \psi(M) /_{c_0} C_0(n) \).

In order to prove undecidability of inclusion we use the family \( P_k (k \in \mathbb{N}) \) of pushdown threads discussed in Example 4:

**Definition 3.** For \( n \in \mathbb{N} \), define \( P_n = P /_{c_0} C_0(n) \) by

\[
P = (c_0.\text{inc} \circ P) \trianglelefteq a \trianglerighteq ((P \trianglelefteq c_0.\text{dec} \trianglerighteq D) \trianglelefteq a \trianglerighteq (D \trianglelefteq c_0.\text{dec} \trianglerighteq P)).
\]
This definition yields the following infinite recursive specification:

\[ \mathcal{P}_0 = \mathcal{P}_1 \preceq a \trianglerighteq (D \preceq a \trianglerighteq \mathcal{P}_0) \]

\[ \mathcal{P}_{k+1} = \mathcal{P}_{k+2} \preceq a \trianglerighteq (\mathcal{P}_k \preceq a \trianglerighteq D) \text{ for } k \in \mathbb{N}. \]

Note that for all \( n \), \( \mathcal{P}_n \sqsubseteq a^\infty \) (see Example 4) and \( \mathcal{P}_n \not\sqsubseteq S \). For future reference we derive the following identities:

\[ \mathcal{P}_0 = [\mathcal{P}_2 \preceq a \trianglerighteq (\mathcal{P}_0 \preceq a \trianglerighteq D)] \preceq a \trianglerighteq (D \preceq a \trianglerighteq \mathcal{P}_0) \]

and for \( m > 0 \),

\[ \mathcal{P}_m = [\mathcal{P}_{m+2} \preceq a \trianglerighteq (\mathcal{P}_m \preceq a \trianglerighteq D)] \preceq a \trianglerighteq (\mathcal{P}_{m-1} \preceq a \trianglerighteq D) \]

We shall prove that for any Minsky thread \( M \),

\[ \frac{M}{\phi_0} C_0(0)/c_1 C_1(0) = S \Leftrightarrow \mathcal{P}_0 \not\sqsubseteq \psi(M)/\phi_0 C_0(0). \]

We first prove this equivalence for Minsky threads that use counters with arbitrary initial values (Corollaries 1 and 2, respectively). To enhance readability we shall write in proofs often

\[ \frac{M}{n/m} \text{ for } \frac{M}{\phi_0} C_0(n)/c_1 C_1(m), \text{ and} \]

\[ \psi(M)/n \text{ for } \psi(M)/\phi_0 C_0(n). \]

**Lemma 1.** Let \( k \in \mathbb{N} \). For all Minsky threads \( M \) and for all \( n, m \in \mathbb{N} \),

\[ \|M, n/m\| = k \Rightarrow \mathcal{P}_m \not\sqsubseteq \psi(M)/\phi_0 C(n). \]

**Proof.** By induction on \( k \) using case ramification, i.e., considering all possible forms of the equation specifying \( M \).

1. \( k = 0 \). If \( \|M, n, m\| = 0 \), then \( M = S \) must be \( M \)'s defining equation and (6) follows immediately.

2. \( k > 0 \). If \( \|M, n, m\| = k \), there are four possible forms for the equation specifying \( M \):
   a) \( M = M_i \preceq c_0.\text{inc} \trianglerighteq M_j \). Then \( \|M_i, n+1, m\| = k-1 \) and
   \[ \psi(M)/n = (a^\infty \preceq a \trianglerighteq c \circ \psi(M_i)/n) \preceq a \trianglerighteq a^\infty. \] (7)
   By induction \( \mathcal{P}_m \not\sqsubseteq \psi(M)/n+1 \). Assume \( \mathcal{P}_m \sqsubseteq \psi(M)/n \). Then by (4) or (5) and (7), \( \mathcal{P}_m \sqsubseteq \psi(M_i)/n+1 \), contradicting the induction hypothesis. Hence \( \mathcal{P}_m \not\sqsubseteq \psi(M)/n \).
   b) \( M = M_i \preceq c_0.\text{dec} \trianglerighteq M_j \). This case is proved similarly, making a case distinction between \( n = 0 \) and \( n > 0 \).
   c) \( M = M_i \preceq c_1.\text{inc} \trianglerighteq M_j \). Then \( \|M_i, n, m+1\| = k-1 \) and
   \[ \psi(M)/n = \psi(M_i)/n \preceq a \trianglerighteq a^\infty. \] (8)

13
By induction $P_{m+1} \not\subseteq \psi(M_i) / n$. Assume $P_m \subseteq \psi(M) / n$. Then by definition of $P_m$ and (8), $P_{m+1} \subseteq \psi(M_i) / n$, contradicting the induction hypothesis. Hence $P_m \not\subseteq \psi(M) / n$.

d) $M = M_i \leq \text{c1.dec} \geq M_j$. Then

$$\psi(M) / n = a^\infty \leq a \geq (\psi(M_i) / n \leq a \geq \psi(M_j) / n).$$ (9)

Let $m = 0$. Then $||M_j, n, 0|| = k-1$ and by induction $P_0 \not\subseteq \psi(M_j) / n$. Assume $P_0 \subseteq \psi(M) / n$. Then by definition of $P_0$ and (9), $P_0 \subseteq \psi(M_j) / n$, contradicting the induction hypothesis. Hence $P_0 \not\subseteq \psi(M) / n$.

Let $m > 0$. Then $||M_i, n, m-1|| = k-1$ and by induction $P_{n-1} \not\subseteq \psi(M_i) / n$. Assume $P_m \subseteq \psi(M) / n$. Then by definition of $P_m$ and (9), $P_{m-1} \subseteq \psi(M_i) / n$, contradicting the induction hypothesis. Hence $P_m \not\subseteq \psi(M) / n$.

Combining (3) and Lemma 1 immediately yields

**Corollary 1.** For all Minsky threads $M$ and for all $n, m \in \mathbb{N}$,

$$M / c_0 C_0(n) / c_1 C_1(m) = S \Rightarrow P_m \not\subseteq \psi(M) / c_0 C_0(n).$$

The next result is about the inclusion of finite approximations.

**Lemma 2.** For all $k \in \mathbb{N}$, Minsky threads $M$ and $n, m \in \mathbb{N}$,

$$M / c_0 C_0(n) / c_1 C_1(m) = D \Rightarrow \pi_k(P_m) \subseteq \pi_k(\psi(M) / c_0 C_0(n)).$$ (10)

**Proof.** By induction on $k$ with base cases $k = 0$ and $k = 1$.

If $k = 0$ then (10) follows trivially.

If $k = 1$ and $M / n / m = D$, then $M = S$ is not $M$’s defining equation. Therefore, $\pi_1(\psi(M) / n) = a \circ D = \pi_1(P_m)$, which proves (10). (Note that already in this case it is essential that $\psi$ introduces $a$-actions in the transformation of $c_0$-equations.)

Assume $k \geq 2$ and $M / n / m = D$. Again, $M = S$ can not be $M$’s defining equation. Consider the remaining four possibilities for $M$’s defining equation:

a) $M = M_i \leq \text{c0.inc} \geq M_j$. Then $M_i / n+1 / m = D$ and

$$\pi_k(\psi(M) / n) = [\pi_k-2(a^\infty) \leq a \geq \pi_k-2(a \circ \psi(M_i) / n+1)] \leq a \geq \pi_k-1(a^\infty).$$

By induction, $\pi_l(P_m) \subseteq \pi_l(\psi(M_l) / n+1)$ for $l < k$. Assume $m = 0$. Then by (4)

$$\pi_k(P_0) = [\pi_k-2(P_2) \leq a \geq \pi_k-2(P_0 \leq a \geq D)] \leq a \geq \pi_k-1(D \leq a \geq P_0) \subseteq [\pi_k-2(a^\infty) \leq a \geq \pi_k-2(a \circ \psi(M_i) / n+1)] \leq a \geq \pi_k-1(a^\infty) = \pi_k(\psi(M) / n).$$

14
Assume \( m > 0 \). Then by (5)

\[
\pi_k(P_m) = [\pi_{k-2}(P_{m+2}) \subseteq a \models \pi_{k-2}(P_m \subseteq a \models D)] \subseteq a \models \pi_{k-1}(P_{m-1} \subseteq a \models D) \\
\subseteq [\pi_{k-2}(a^\infty) \subseteq a \models \pi_{k-2}(a \circ \psi(M_i)/n+1)] \subseteq a \models \pi_{k-1}(a^\infty) \\
= \pi_k(\psi(M)/n).
\]

b) \( M = M_i \triangleleft \text{c0.dec} \triangleright M_j \). Assume \( n = 0 \). Then \( M_j/0/m = D \) and

\[
\pi_k(\psi(M)/0) = [\pi_{k-2}(a^\infty) \subseteq a \models \pi_{k-2}(a \circ \psi(M_j)/0)] \subseteq a \models \pi_{k-1}(a^\infty).
\]

By induction, \( \pi_l(P_m) \subseteq \pi_l(\psi(M_j)/0) \) for \( l < k \). As in case a), it follows that both for \( m = 0 \) and \( m > 0 \),

\[
\pi_k(P_m) \subseteq [\pi_{k-2}(a^\infty) \subseteq a \models \pi_{k-2}(a \circ \psi(M_j)/0)] \subseteq a \models \pi_{k-1}(a^\infty) \\
= \pi_k(\psi(M)/n).
\]

The case \( n > 0 \) is proved similarly.

c) \( M = M_i \triangleleft \text{c1.inc} \triangleright M_j \). Then \( M_i/n/m+1 = D \) and

\[
\pi_k(\psi(M)/n) = \pi_{k-1}(\psi(M_i)/n) \subseteq a \models \pi_{k-1}(a^\infty).
\]

By induction, \( \pi_l(P_{m+1}) \subseteq \pi_l(\psi(M_i)/0) \) for \( l < k \). Assume \( m = 0 \). Then

\[
\pi_k(P_0) = \pi_{k-1}(P_1) \subseteq a \models \pi_{k-1}(D \subseteq a \models P_0) \\
\subseteq \pi_{k-1}(\psi(M_i)/n) \subseteq a \models \pi_{k-1}(a^\infty) \\
= \pi_k(\psi(M)/n).
\]

Assume \( m > 0 \). Then

\[
\pi_k(P_m) = \pi_{k-1}(P_{m+1}) \subseteq a \models \pi_{k-1}(P_{m-1} \subseteq a \models D) \\
\subseteq \pi_{k-1}(\psi(M_i)/n) \subseteq a \models \pi_{k-1}(a^\infty) \\
= \pi_k(\psi(M)/n).
\]

d) \( M = M_i \triangleleft \text{c1.dec} \triangleright M_j \). This case can be proved in a similar style, again making a case distinction between \( m = 0 \) and \( m > 0 \).

In the proof above, the cases a) and b) clearly motivate \( \psi \)'s definition on \( \text{c0-terms} \): the simulation of a \( \text{c1.inc} \circ \text{c1.dec-prefix} \) generates in the case of divergence on \( C_0 \) the thread \( a^\infty \), which is needed to include \( P_m \). Lemma 2 immediately extends to the inclusion of infinite threads:

**Corollary 2.** For all Minsky threads \( M \) and for all \( n, m \in \mathbb{N} \),

\[
M/\omega_0 C_0(n)/\omega_1 C_1 (m) = D \Rightarrow P_m \subseteq \psi(M)/\omega_0 C_0(n).
\]

Corollary 1 and Corollary 2 connect the halting problem for Minsky machines and inclusion between certain pushdown threads:
Corollary 3. Let $M$ be a Minsky thread. Then

$$M/c_0C_0(0)/c_1 C_1(0) = S \iff \mathcal{P}_0 \not\subseteq \psi(M)/c_0 C_0(0).$$

Since counters can be seen as particular stacks, we can by Corollary 3 summarize the reduction from the halting problem for Minsky machines to

Theorem 4. It is undecidable whether for a stack $S$ and regular threads $P$ and $Q$ it holds that

$$P/a S(\alpha) \subseteq Q/a S(\beta).$$

5 Programming regular threads

*Program Algebra* [5], or PGA for short, is a program notation for regular threads. On PGA a hierarchy of program notations is founded, comprising languages that contain more and more sophisticated programming features, but that are all equally expressive. We will show that PGA exactly describes the class of regular threads.

Based on a finite set $A$ of basic instructions, PGA has the binary operators “concatenation” and “repetition”, and five kinds of instructions:

- **Concatenation**, notation $\cdot \omega$. If $X$ and $Y$ are PGA-terms, so is $X;Y$.
- **Repetition**, notation $(\cdot)^\omega$. If $X$ is a PGA-term, so is $X^\omega$.

**Basic instruction** $a \in A$. 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 (i.e., the thread D) will occur.

- **Positive test instruction** $+a$ for $a \in A$. The 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. If there is no subsequent instruction to be executed, inaction occurs.

- **Negative test instruction** $-a$ for $a \in A$. The instruction $-a$ executes like the basic instruction $a$. Upon true, the program skips its next instruction and continues with the instruction 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 (it defines the thread $S$).

- **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.
Instruction sequence congruence for PGA-terms is axiomatized by the axioms PGA1-4 in Table 1. Here PGA2 is an axiom-scheme that is parametric in $n$ where in PGA, $X^1 = X$ and $X^{k+1} = X; X^k$.

<table>
<thead>
<tr>
<th>$(X; Y); Z = X; (Y; Z)$</th>
<th>(PGA1)</th>
<th>$X^\omega; Y = X^\omega$</th>
<th>(PGA3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(X^n)^\omega = X^\omega$ for $n &gt; 0$</td>
<td>(PGA2)</td>
<td>$(X; Y)^\omega = X; (Y; X)^\omega$</td>
<td>(PGA4)</td>
</tr>
</tbody>
</table>

**Table 1.** Axioms for PGA’s instruction sequence congruence

With the axioms PGA1-4 one easily derives unfolding, i.e., $X^\omega = X; X^\omega$.

Each closed PGA-term is considered a program of which the behaviour is a regular thread in $\text{BTA}^\omega$, viewing $A$ as the set of actions. The thread extraction operator $[X]$ assigns a thread to a closed term $X$. Thread extraction is defined by the thirteen equations in Table 2, where $a \in A$ and $u$ is an arbitrary instruction.

| $\| | = S$ | $\| ; X | = S$ | $\# | = D$ | $\# | = D$ |
|------------|-------------|----------|----------|
| $| a | = a \circ D$ | $| a ; X | = a \circ | X |$ | $| 0 | = D$ | $| a ; X | = a \circ | X |$ |
| $| + a | = a \circ D$ | $| + a ; X | = | X | \leq a \triangleright | # 2 ; X |$ | $| 1 | = X$ | $| # 2 ; X | \leq a \triangleright | X |$ |
| $| a | = a \circ D$ | $| - a ; X | = | # 2 ; X | \leq a \triangleright | X |$ | $| # k + 2 ; u | = D$ | $| # k + 2 ; u ; X | = | # k + 1 ; X |$

**Table 2.** Equations for thread extraction on PGA

Two examples:

1. $| (0)^\omega | = | 0 ; (0)^\omega | = D,$
2. $| - a ; b ; c | = | # 2 ; b ; c | \leq a \triangleright | b ; c |$
   $= | # 1 ; c | \leq a \triangleright b \circ | c |$
   $= | c | \leq a \triangleright b \circ c \circ D$
   $= c \circ D \leq a \triangleright b \circ c \circ D.$

In some cases, the equations in Table 2 can be applied from left to right yielding equations of the form $| X | = | Y | = \ldots = | X |$ and no behaviour, e.g.,

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

In such cases, the extracted thread is defined as $D$. 

17
In all other cases, thread extraction yields a finite (linear) recursive specification, and thus a regular thread.

**Example 5.** Let \( P = [(a;+b;\#3;−b;\#4)^\omega]. \) Then
\[
P = a \circ Q
\]
\[
Q = P \leq b \supset R
\]
\[
R = P \leq b \supset R.
\]

From the above considerations it is clear that any PGA-program defines a regular thread. Conversely, each regular thread can be described as the thread extraction of a PGA term. We here only sketch why this is the case. First \( S = [!] \) and \( D = [\#0]. \) Without loss of generality one can assume that every other thread \( P \) satisfies \( \pi_1(P) = a \circ D \) for some \( a, \) and solves \( X_1 \) in a finite linear recursive specification \( X_1 = X_i \leq a \supset X_j \) in which \( S \) and \( D \) occur only in the last two equations. In Example 5, \( P \) is a fixed point for \( X_1 \) in the tailored specification
\[
X_1 = X_2 \leq a \supset X_2
\]
\[
X_2 = X_1 \leq b \supset X_3
\]
\[
X_3 = X_1 \leq b \supset X_3
\]
\[
X_4 = S
\]
\[
X_5 = D.
\]
(Note that each fixed point for \( X_1 \) through \( X_3 \) does not refer to the equations for \( X_4 \) and \( X_5. \)) One can transform any such tailored specification to a program of the form \((u_1;\ldots;u_k)^\omega\) in the following way:
\[
X_1 = X_{1,l} \leq a_1 \supset X_{1,r} \quad \Rightarrow \quad (+a_1;\#c(1,l);\#c(1,r);
\]
\[
\vdots
\]
\[
X_i = X_{i,l} \leq a_i \supset X_{i,r} \quad +a_i;\#c(i,l);\#c(i,r);
\]
\[
\vdots
\]
\[
X_n = S \quad !
\]
\[
X_{n+1} = D \quad \#0)^\omega.
\]

If each value \( c(i,j) \) is chosen such that the jump is to the position in the program that matches the transformation of \( X_{i,l} = t_{i,l}(X), \) thread extraction yields the same recursive specification. Returning to Example 5,
\[
X_1 = X_2 \leq a \supset X_2 \quad \Rightarrow \quad (+a;\#2;\#1;
\]
\[
X_2 = X_1 \leq b \supset X_3 \quad +b;\#7;\#1;
\]
\[
X_3 = X_1 \leq b \supset X_3 \quad +b;\#4;\#9;
\]
\[
X_4 = S \quad !
\]
\[
X_5 = D \quad \#0)^\omega.
\]

Clearly, \(|(a;+b;\#3;−b;\#4)^\omega| = |(+a;\#2;\#1;+b;\#7;\#1;+b;\#4;\#9;!;\#0)^\omega|. \]
6 Conclusions

Pushdown threads can be used in program algebra based semantics of sequential or object-oriented programming, for instance as described in [3]. In that approach, a single stack is used to store the arguments of a method call. Furthermore, pushdown threads are important for the theoretical foundation of program algebra itself, for instance admitting easy definitions of programming notations in which recursion can be expressed. This explains our interest in the decidability result proved in Section 3.

The undecidability of inclusion for pushdown threads is proved using a construction in which one of the counters is “weakly simulated”. This method of Jančar is recorded first in 1994 [12], where it is used to prove various undecidability results for Petri nets. In 1999, Jančar et al. [13] used the same idea to prove the undecidability of simulation preorder for processes generated by one-counter machines, and this is most comparable to our approach. However, in the case of pushdown threads the inclusion relation itself is a little more complex than in process simulation or language theory because \( D \subseteq P \) for any thread \( P \). Moreover, threads have restricted branching, and therefore transforming a regular (control) thread into one that simulates one of the counters of a Minsky machine is more complex than in the related approaches referred to above. Also, the particular thread \( P_0 \) used to prove our undecidability result (Corollary 3) has to be much more structured than the related nets/processes in [12, 13].

References

Within this series the following reports appeared.


The above reports and more are available through the website: www.science.uva.nl/research/prog/