On the Computational Strength of Pure Ambient Calculi [1]

Sergio Maffeis, Iain Phillips [*]

Department of Computing
Imperial College London
London SW7 2AZ, England

Abstract

Cardelli and Gordon's calculus of Mobile Ambients has attracted widespread interest as a model of mobile computation. The standard calculus is quite rich, with a variety of operators, together with capabilities for entering, leaving and dissolving ambients. The question arises of what is a minimal Turing-complete set of constructs. Previous work has established that Turing completeness can be achieved without using communication or restriction. We show that it can be achieved merely using movement capabilities (and not dissolution). We also show that certain smaller sets of constructs are either terminating or have decidable termination.

Key words: Ambient calculus, counter machine, Turing completeness, decidability, termination

1 Introduction

Since its introduction in 1998, Cardelli and Gordon's calculus of Mobile Ambients (MA) [9] has attracted widespread interest as a model of mobile computation. An *ambient* is a vessel containing running processes. Ambients can move, carrying their contents with them. The standard calculus is quite rich, with a variety of operators, together with capabilities for entering, leaving and dissolving ambients. Subsequent researchers have increased this variety by proposing alternative movement capabilities. We may mention Mobile Safe

[*] Corresponding author. Tel:+44 (0)20 7594 8265 Fax:+44 (0)20 7581 8024
Email addresses: `maffeis@doc.ic.ac.uk` (Sergio Maffeis), `iccp@doc.ic.ac.uk` (Iain Phillips).

[1] A shorter version of this paper appeared in EXPRESS 2003 [16].

Ambients (SA) [15], Robust Ambients (ROAM) [13], Safe Ambients with Passwords (SAP) [17], the Push and Pull Ambient Calculus (PAC) [22], Controlled Ambients (CA) [27], and the version of Boxed Ambients (BA) [3] with passwords (NBA) [5]. We shall use the term Ambient Calculus (AC) to refer to all of these variants.

The question arises of what is a minimal set of constructs which gives the computational power of Turing machines, i.e. is *Turing-complete.* One way to tackle this is to encode into the Ambient Calculus some other process calculus which is known to be Turing-complete. Cardelli and Gordon showed how to encode the asynchronous π-calculus into MA [9]. The encoding makes use of MA's communication primitives. However Cardelli and Gordon also encoded Turing machines directly into *pure* MA, where there is no communication. (Incidentally, Zimmer [28] subsequently encoded the synchronous π-calculus without choice into pure SA.)

Busi and Zavattaro [8] showed how to encode counter machines into pure public MA (where by "public" we mean lacking the restriction operator). Independently, Hirschkoff, Lozes and Sangiorgi [14] encoded Turing machines into the same sub-calculus. In this paper we follow up this work and investigate whether even smaller fragments of AC can be Turing-complete. We concentrate entirely on pure AC. Our work is very much inspired by that of Busi and Zavattaro; we follow them in using counter machines rather than Turing machines.

The major question left open by previous work is whether pure AC without the open capability which dissolves ambients can be Turing-complete. This question is of particular interest in view of the decision which Bugliesi, Castagna and Crafa took to dispense with ambient opening when proposing their calculus of Boxed Ambients (BA) [3,18,5,10]. They advocate communication between ambients where one is contained in the other, rather than the same-ambient communication of MA. A similar model of communication is employed in [21].

We give an encoding of counter machines into pure public MA without the open capability (Theorem 3.10), showing that this fragment is Turing-complete. The encoding also demonstrates that both termination and the observation of weak barbs are undecidable problems. As far as we are aware, Turing completeness has not previously been shown for any pure ambient calculus without the capability to dissolve ambients (although we note that an encoding of π-calculus into BA with communication is given in [3]).

Two different kinds of ambient movement were identified by Cardelli and Gordon [9]: subjective and objective. *Subjective* movement is where an ambient moves itself; *objective* movement is where it is moved by another ambient.

For instance, if $m[\,P\,]$ (an ambient named m containing process P) is to enter another ambient $n[\,Q\,]$, then control can reside in P or in Q. The standard calculus MA opts for subjective movement, while objective movement (so-called "push and pull") has been studied in [22]. We shall show that counter machines can be encoded into the pure push and pull calculus (PAC) without the open capability.

A number of calculi are hybrids between subjective and objective movement: when handling the entry of $m[\,P\,]$ into $n[\,Q\,]$, they require P and Q to synchronise. In Mobile Safe Ambients (SA) [15], an ambient must explicitly allow itself to be entered by means of a *co-capability*. It is straightforward to encode standard MA into SA by equipping each ambient with the necessary co-capabilities. Therefore Turing completeness results for MA, such as that mentioned above, will extend to SA, but not the other way round.

Robust Ambients (ROAM) [13] is another calculus where ambients must synchronise to perform an entry. For $m[\,P\,]$ to enter $n[\,Q\,]$, P must name n and Q must name m, which is a symmetrical blending of subjective and objective movement. Turing completeness results for either MA or PAC will extend to ROAM (since our encodings use only a finite set of names).

As remarked above, MA and PAC are less synchronised *between* ambients than SA or ROAM. Movement can be made less synchronous *within* ambients if we require that movement capabilities have no continuations, so that if $m[\,P\,]$ enters $n[\,Q\,]$ then neither P nor Q can rely on when this has happened in the rest of their code. This may be called *asynchronous* movement. We show that both subjective and objective calculi with asynchronous movement (and without restriction) are Turing-complete—there is enough power in processes being able to synchronise on dissolving ambients.

As far as infinite behaviour is concerned, ambients are usually endowed with the replication operator, and our main results focus on variants of the ambient calculus with this operator. Nonetheless, Busi and Zavattaro have shown that the strikingly simple sub-calculus having only the open capability and empty ambients, but with restriction and recursion, is Turing-complete. For completeness, we show that the same is true for a calculus having only the push capability of PAC. Unlike in the π-calculus case, where recursion or replication are inter-definable, having one or the other in the ambient calculus has a significant impact.

We are interested in finding *minimal* Turing-complete fragments of AC. This entails showing that smaller fragments are too weak to be Turing-complete. Busi and Zavattaro have shown that in the fragment of pure MA with the open capability, but without movement capabilities, it is decidable whether a given process has a non-terminating computation [8]. We show the same decidability property for public fragments with capabilities allowing movement in one direction only (either entering or exiting). We also show that in certain smaller fragments (where replication is only allowed on capabilities) every computation terminates.

In this paper we focus on the *computational strength* of fragments of the ambient calculus, rather than their relative expressiveness, and therefore we do not investigate whether different fragments (e.g. with synchronous or asynchronous movement) are mutually encodable.

Figure 1 illustrates the main results of this paper for MA and BA. The arrows represent inclusions.

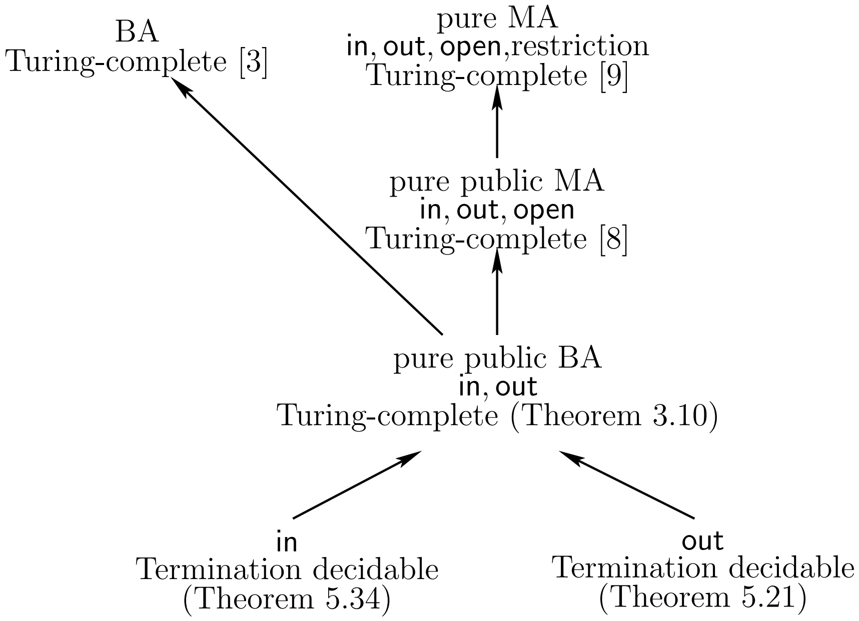

Fig. 1. Main results for MA and BA

Figure 2 illustrates the main results of this paper for PAC.

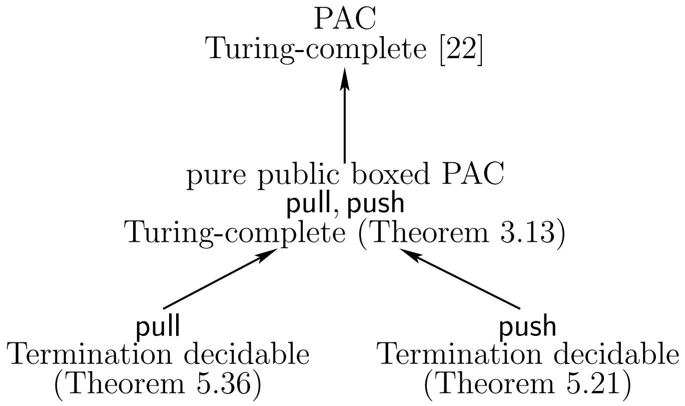

Fig. 2. Main results for PAC

The paper is organised as follows. In Section 2 we recall various operators and capabilities of the Ambient Calculus, together with their associated notions of reduction. In Section 3 we discuss various Turing-complete languages, with and without the open capability. In Section 4 we show that certain fragments of AC with replication are in fact terminating. In Section 5 we show that certain other fragments of AC have decidable termination. Finally we draw some conclusions.

1.1 Related Work

In independent work, Boneva and Talbot [2] present an encoding of two-counter machines (a Turing-complete formalism) into pure public BA. The fragment of MA we consider in Theorem 3.10 is similar to theirs, but they allow replication on arbitrary processes, while we only allow replication on capabilities. They show that reachability and name convergence (the observation of weak barbs) are both undecidable problems. As their encoding can take "wrong turnings" and is divergent, they have left the Turing completeness of their fragment of MA as an open question. We show Turing completeness for our fragment, and as a corollary we obtain the undecidability of termination and of name convergence. Our methods do not show that reachability is undecidable, while their methods do not show that termination is undecidable.

The focus of our work is different from that of Boneva and Talbot, in that we concentrate on Turing completeness and termination, while they concentrate on reachability and model-checking in the ambient logic.

2 Operators and Capabilities

We will investigate a variety of operators and capabilities of pure Mobile Ambients (MA) [9] and variants thereof. We let $P, Q, \ldots$ range over (process) terms and $M, \ldots$ over capabilities which can be exercised by ambients. We assume a set $\mathcal{N}$ of names, ranged over by $m, n, \ldots$, and a set of process variables (used for recursion), ranged over by $X, \ldots$.

First we state a "portmanteau" language of (process) terms which contains all the *operators* which we shall consider.

$$P ::= \mathbf{0} \mid n[\,P\,] \mid P \,|\, Q \mid M.P \mid \nu n\, P \mid\, !\,P \mid X \mid \mathsf{rec}\, X.P$$

Here as usual $\mathbf{0}$ denotes the inactive process. We shall feel free to omit trailing $\mathbf{0}$s and write empty ambients as $n[\,]$ rather than $n[\,\mathbf{0}\,]$. The term $n[\,P\,]$ is an ambient named n containing term P. The term $P \mid Q$ is the parallel composition of P and Q. We write P^i for the parallel composition of i copies of P (the laws of structural congruence stated below will ensure that parallel composition is associative). The term $M.P$ performs capability M and then continues with P. The term $\nu n\, P$ is term P with name n restricted. As usual, restriction is a name-binding operator. We denote the set of free names of a term P by $\mathsf{fn}(P)$. The term $!\, P$ is a replicated term which can spin off copies of P as required. The term $\mathsf{rec}\ X.P$ is a recursion in which X is a bound process variable. We shall call terms with no free process variables "processes" (the closed terms). We shall refer to "terms" when we mean terms possibly with free process variables (i.e. open terms). Recursion is *unboxed* [24,8] if in $\mathsf{rec}\, X.P$ any occurrence of X within P is not inside an ambient. We shall only require unboxed recursion. If recursion is available then $!\, P$ can be simulated by $\mathsf{rec}\, X.(X \mid P)$, and so we shall never require both replication and recursion.

Here is the set of all *capabilities* we shall consider:

$$M ::= \mathsf{open}\, n \mid \overline{\mathsf{open}}\, n \mid \mathsf{in}\, n \mid \overline{\mathsf{in}}\, n \mid \mathsf{out}\, n \mid \overline{\mathsf{out}}\, n \mid \mathsf{push}\, n \mid \mathsf{pull}\, n$$

The first capability $\mathsf{open}\, n$ is used to dissolve an ambient named n. Sometimes we consider the "safe" version [15] where the ambient being opened performs "co-capability" $\overline{\mathsf{open}}\, n$. The remaining capabilities all relate to movement. We can distinguish between *subjective* and *objective* moves: The capabilities $\mathsf{in}\, n$ and $\mathsf{out}\, n$ enable an ambient to enter or leave an ambient named n. This is subjective movement. Again, sometimes we consider the "safe" versions of the capabilities where the ambient being entered or left performs "co-capabilities" $\overline{\mathsf{in}}\, n$ or $\overline{\mathsf{out}}\, n$. By contrast, objective movement is where ambients are moved by fellow ambients. We consider the so-called "push" and "pull" capabilities of PAC [22]. An ambient containing another ambient named n can use the capability $\mathsf{push}\, n$ to push the other ambient out. Similarly $\mathsf{pull}\, n$ can be used to pull in an ambient named n.

Capabilities act as "guards", in the sense that given a term $M.P$, capability M must be consumed before P becomes active. We shall say that an occurrence of P in Q is *guarded* if P is a subterm of some subterm $M.R$ of Q.

Structural congruence $\equiv$ equates terms which are the same up to structural rearrangement. It is defined to be the least congruence satisfying the following rules:

$$\begin{array}{rcl}
\mathbf{0} \mid P \equiv P & \qquad & \nu n\, \mathbf{0} \equiv \mathbf{0} \\
P \mid Q \equiv Q \mid P & & \nu m\, \nu n\, P \equiv \nu n\, \nu m\, P \\
(P \mid Q) \mid R \equiv P \mid (Q \mid R) & & !\,P \equiv P \mid !\,P \\
\nu n\,(P \mid Q) \equiv (\nu n\, P) \mid Q \text{ if } n \notin \mathsf{fn}(Q) & & \mathsf{rec}\, X.P \equiv P\{\mathsf{rec}\, X.P/X\} \\
\nu n\, m[\,P\,] \equiv m[\,\nu n\, P\,] \text{ if } m \neq n & &
\end{array}$$

When we say that computation is deterministic (when discussing encodings of counter machines into various ambient languages), we identify structurally congruent processes.

On several occasions we shall make use of *commutative-associative structural congruence* $\equiv_{ca}$, which is the least congruence satisfying the laws:

$$P \mid Q \equiv_{ca} Q \mid P \qquad (P \mid Q) \mid R \equiv_{ca} P \mid (Q \mid R)$$

This has the property that for any term P the set $\{Q : Q \equiv_{ca} P\}$ is finite.

The *reduction* relation $\rightarrow$ between processes describes how one process can evolve to another in a single step. We start by defining the reductions associated with the capabilities.

$$\begin{array}{lr}
\text{(Open)} & \mathsf{open}\, n.P \mid n[\,Q\,] \rightarrow P \mid Q \\
\text{(In)} & n[\,\mathsf{in}\, m.P \mid Q\,] \mid m[\,R\,] \rightarrow m[\,n[\,P \mid Q\,] \mid R\,] \\
\text{(Out)} & m[\,n[\,\mathsf{out}\, m.P \mid Q\,] \mid R\,] \rightarrow n[\,P \mid Q\,] \mid m[\,R\,] \\
\text{(SafeOpen)} & \mathsf{open}\, n.P \mid n[\,\overline{\mathsf{open}}\, n.Q \mid R\,] \rightarrow P \mid Q \mid R \\
\text{(SafeIn)} & n[\,\mathsf{in}\, m.P \mid Q\,] \mid m[\,\overline{\mathsf{in}}\, m.R \mid S\,] \rightarrow m[\,n[\,P \mid Q\,] \mid R \mid S\,] \\
\text{(SafeOut)} & m[\,n[\,\mathsf{out}\, m.P \mid Q\,] \mid \overline{\mathsf{out}}\, m.R \mid S\,] \rightarrow n[\,P \mid Q\,] \mid m[\,R \mid S\,] \\
\text{(Pull)} & n[\,\mathsf{pull}\, m.P \mid Q\,] \mid m[\,R\,] \rightarrow n[\,P \mid Q \mid m[\,R\,]\,] \\
\text{(Push)} & n[\,m[\,P\,] \mid \mathsf{push}\, m.Q \mid R\,] \rightarrow n[\,Q \mid R\,] \mid m[\,P\,]
\end{array}$$

We shall be considering languages which only possess a subset of the full set of capabilities. When we consider languages with capability $\overline{\mathsf{open}}$, we shall always have capability open as well, and we shall adopt rule (SafeOpen) and not rule (Open). Clearly, if a language has capabilities open, $\overline{\mathsf{open}}$ and replication on these capabilities, then the effect of rule (Open) can be simulated: every ambient can be made perfectly receptive to being opened by converting $n[\,P\,]$

into $n[\,!\,\overline{\mathsf{open}}\,n \mid P\,]$. Similar considerations apply to capabilities $\overline{\mathsf{in}}$ and in, $\overline{\mathsf{out}}$ and out.

The remaining rules for reduction are

$$\text{(Amb)}\ \frac{P \to P'}{n[\,P\,] \to n[\,P'\,]} \qquad \text{(Par)}\ \frac{P \to P'}{P \mid Q \to P' \mid Q}$$

$$\text{(Res)}\ \frac{P \to P'}{\nu n\, P \to \nu n\, P'} \qquad \text{(Str)}\ \frac{P \equiv P' \quad P' \to Q' \quad Q' \equiv Q}{P \to Q}$$

We write $\Rightarrow$ for the reflexive and transitive closure of $\to$.

A *language* is a pair $(L, \to)$ consisting of a set of processes L together with a reduction relation $\to$. We shall write $(L, \to)$ as L for short. We let $L, \ldots$ range over languages. We shall define a language by giving the set of processes. The reduction relation (and structural congruence) for the language will be tacitly assumed to be given by the set of all the rules in this section which are applicable to the available operators and capabilities, except as noted above for the "safe" and standard versions of the in and out capabilities. A *computation* is a maximal sequence of reductions $P_0 \to P_1 \to \cdots$.

The most basic observation that can be made of a process is the presence of top-level ambients (i.e. unguarded ambients which are not contained in other ambients) [9]. We say that n is a *strong barb* of P ($P \downarrow n$) iff $P \equiv \nu m_1 \ldots m_k\,(n[\,Q\,] \mid R)$ for some Q and R (where $n \neq m_1, \ldots, m_k$), and n is a *weak barb* of P ($P \Downarrow n$) iff $P \Rightarrow\downarrow n$.

3 Turing-complete Fragments of AC

A basic measure of the computational strength of a process language is whether Turing machines, or some other Turing-complete formalism, can be encoded in the language. Cardelli and Gordon [9] established that pure MA can encode Turing machines. Busi and Zavattaro [8] improved this result by showing that counter machines (CMs) can be encoded in pure public MA.

We shall show that CMs can be encoded in pure public MA without open, which can be called pure public BA. We shall also encode CMs in a version of MA with asynchronous movement (i.e. no continuations after capabilities), but with the open capability.

A *Counter Machine (CM)* is a finite set of registers $R_0, \ldots, R_b$ ($b \in \mathbb{N}$). Each R_j contains a natural number. We write $R_j(k)$ for R_j together with its contents

k. Initially the registers hold the input values. The CM executes a numbered list of instructions $I_0, \ldots, I_a$ ($a \in \mathbb{N}$), where I_i is of two forms:

- $i : \mathsf{Inc}(j)$ adds one to the contents of R_j, after which control moves to I_{i+1}.
- $i : \mathsf{DecJump}(j, i')$ subtracts one from the contents of R_j, after which control moves to I_{i+1}, unless the contents are zero, in which case R_j is unchanged and the CM jumps to instruction i'.

The CM starts with instruction I_0, and executes instructions in sequence indefinitely, until control moves to an invalid instruction number (which we can take to be $a + 1$), at which point the CM terminates, and the output is held in the first register.

CMs as defined above are basically the Unlimited Register Machines of [26]. They use a set of instructions which is minimal while retaining Turing completeness [20]. (In fact CMs with no more than two registers are already Turing-complete.)

3.1 Criteria for Turing Completeness

It is best to make clear what criterion for Turing completeness we shall use in this paper.

In the classical setting, a programming language is *Turing-complete* if for every partial recursive function (equivalently, every CM-computable function) there is a program in the language which computes it. It is understood that whenever the recursive function is defined, the corresponding program is *guaranteed* to yield the correct value (and not to fail to complete, or to give wrong results).

We now consider what this might mean in the setting of process calculi, and ambient calculi in particular. Let CM be a CM (program plus registers with their contents). Let $[\![CM]\!]$ be the encoding of CM in a target fragment of AC. We shall require the following:

Criterion 3.1 • *If CM terminates then* every *computation of $[\![CM]\!]$ completes successfully, meaning that it signals completion to other processes in some manner, obtains the correct result and makes the result of the computation (i.e. the contents of the first register) available in usable form to potential subsequent computations to be performed by other processes.*

- *If CM does not terminate, then* no *computation of $[\![CM]\!]$ signals completion.*

Notice that this criterion offers a *guarantee* that the CM will be simulated correctly, much as any conventional Turing-complete programming language is guaranteed to compute any partial recursive function.

The two requirements that completion is signalled to other processes and that the result is available to other processes mean that the output is made fully explicit, and that we can sequentially compose encodings of CMs in a straightforward manner. Again, this is what we would expect in any conventional setting; fundamental results such as the undecidability of the halting problem depend on being able to compose machines sequentially.

In our encodings, completion will be signalled by the appearance of a particular ambient at the top level. So we can deduce from the undecidability of the halting problem for CMs that for the target fragment it is undecidable in general for a process P and name n whether $P \Downarrow n$.

Although all our encodings in this paper will satisfy Criterion 3.1, there has been recent interest in weaker notions of Turing completeness, where success of the encoded computation is possible but not guaranteed. Here is a possible formulation of what might be referred to as "may" (as distinct from "must") Turing completeness:

Criterion 3.2 • *If CM terminates then* some *computation of $[\![CM]\!]$ signals completion to other processes. Moreover, if any computation of $[\![CM]\!]$ does signal completion then it does indeed complete successfully, meaning that it obtains the correct result and makes the result of the computation available in usable form to potential subsequent computations to be performed by other processes.*

- *If CM does not terminate, then* no *computation of $[\![CM]\!]$ signals completion.*

Again we have required that output is made fully explicit, allowing sequential composition of encoded CMs. Again also it is likely that completion will be signalled by some kind of barb, the existence of which will therefore be undecidable.

An example satisfying Criterion 3.2 is the encoding by Hirschkoff, Lozes and Sangiorgi [14] of TMs into a fragment of MA, where the encoding may take a "wrong turning". Such wrong turnings are strictly limited, in that the process will halt immediately in a state which cannot be mistaken for successful completion. Since we give in this paper an encoding satisfying Criterion 3.1 into a similar fragment, this does not provide an example of where Criterion 3.2 can be met, but not Criterion 3.1.

A particularly interesting recent result, involving CCS [19] rather than ambient calculi, is the encoding by Busi, Gabbrielli and Zavattaro [7] of CMs into CCS with replication rather than recursion, denoted CCS$_!$. As with the Hirschkoff et al. encoding, processes may take wrong turnings and so there is no guarantee of success. The successful computation will terminate, while faulty computations are forced to diverge. So the encoding satisfies the following criterion:

Criterion 3.3 • *If CM terminates then* some *computation of* $[\![CM]\!]$ *terminates. Moreover, if any computation of* $[\![CM]\!]$ *does terminate then it obtains the correct result, which is available to subsequent computations.*
• *If CM does not terminate, then* no *computation of* $[\![CM]\!]$ *terminates.*

However the encoding does not satisfy Criterion 3.2, since there is no unambiguous signal of completion. The successful computation will produce a barb to indicate that the last instruction has been reached, but faulty computations can also produce this barb, so that it can be misleading to other processes. In fact Busi et al. show that the existence of weak barbs is decidable for $\mathrm{CCS}_!$, so that there is little prospect of satisfying Criterion 3.2. The point is that termination is the only foolproof indication that a computation has completed successfully, and this is not something that can be recognised by CCS. Of course, matters would be different if one moved to a process language where termination can be detected, such as ACP [1]. In fact, if termination can be signalled to other processes then Criterion 3.3 is equivalent to Criterion 3.2.

As stated earlier, our encodings will satisfy Criterion 3.1. They will also satisfy the following additional property:

Criterion 3.4 • *If CM terminates then* every *computation of* $[\![CM]\!]$ *terminates.*
• *If CM does not terminate, then* no *computation of* $[\![CM]\!]$ *terminates.*

We can therefore deduce that it is undecidable whether a process has an infinite computation. (In fact, this can still be deduced if the second item is weakened to: if CM does not terminate, then $[\![CM]\!]$ has an infinite computation.)

However, since Criterion 3.4 is not required for Turing completeness, we cannot deduce that a language fails to be Turing-complete simply because termination is decidable. There could still be an encoding of CMs into the target language where all computations of encoded CMs are infinite. When the CM terminates, the encoded CM reports a result in a finite time before diverging. Despite this, it is possible to achieve separation results by showing Criterion 3.4 for one fragment and decidability of termination for another fragment.

Observe that, unlike Criterion 3.4, Criterion 3.3 does not imply that it is undecidable whether a process has an infinite computation. In fact Busi et al. show that this is decidable for $\mathrm{CCS}_!$ [6]. On the other hand, both criteria imply that the existence of a *finite* computation (*convergence*) is undecidable, as Busi et al. state for $\mathrm{CCS}_!$ [7].

Many encodings (such as the one by Hirschkoff et al. referred to above) satisfy the following one-step preservation property: if CM moves in one step to CM' then $[\![CM]\!] \Rightarrow [\![CM']\!]$. While one-step preservation is useful, we contend that it is needlessly strong for Turing completeness. Consider for instance a Turing machine (TM) which is non-erasing in the following sense: at each step it copies the tape contents to the next unused part of the tape and then makes the change required by the instruction. Such a machine is clearly as powerful as a normal TM. However we cannot encode TMs into non-erasing TMs and satisfy the one-step preservation property, since the non-erasing TM has extra information. (Note that reachability of configurations is decidable for non-erasing TMs, since the tape contents keep on increasing in size, so that Turing completeness does not imply that reachability is undecidable.)

This is relevant to our concerns, since in our encodings we accumulate inert garbage. Just as with non-erasing TMs, this is no barrier to Turing completeness.

3.2 Existing Work

Busi and Zavattaro gave encodings of CMs into two fragments of pure AC. Both encodings are deterministic (up to structural congruence) and satisfy Criteria 3.1 and 3.4. The first fragment, which we shall call L_{ν}^{op}, is defined by

$$P ::= \mathbf{0} \mid n[\,] \mid P \mid Q \mid \mathsf{open}\, n.P \mid \nu n\, P \mid X \mid \mathsf{rec}\, X.P$$

Theorem 3.5 ([8]) *L_{ν}^{op} is Turing-complete.*

It is striking that empty ambients with no movement capabilities are enough. There is an essential use of restriction to obtain the effect of mutual recursion. We shall show that a similar result holds when we substitute push for open (Section 3.3).

Busi and Zavattaro's second encoding of CMs is into the following language, which we shall call $L_{\mathsf{io}}^{\mathsf{op}}$:

$$P ::= \mathbf{0} \mid n[P] \mid P \mid Q \mid \mathsf{open}\, n.P \mid \mathsf{in}\, n.P \mid \mathsf{out}\, n.P \mid !\, P$$

Notice that $L_{\mathsf{io}}^{\mathsf{op}}$ does not require restriction, and uses replication rather than recursion. Clearly, $L_{\mathsf{io}}^{\mathsf{op}}$ is exactly pure public MA.

Theorem 3.6 ([8]) *$L_{\mathsf{io}}^{\mathsf{op}}$ is Turing-complete.*

Independently, Hirschkoff, Lozes and Sangiorgi [14] have encoded Turing machines into $L_{\mathsf{io}}^{\mathsf{op}}$, with the additional syntactic constraint that the continuation of a capability must be *finite*, that is, must not involve replication. As stated above (Section 3.1), this establishes a form of Turing completeness which accords with Criterion 3.2 (“may”), rather than Criterion 3.1 (“must”).

We shall show that Theorem 3.6 can be improved in two ways: the continuations of in and out can be removed (Section 3.4), or the open capability can be removed (Section 3.5).

3.3 Recursion and push

Let L_{rp} be the following language (a fragment of PAC [22], except that we use recursion instead of replication):

$$P ::= \mathbf{0} \mid n[\,P\,] \mid P \mid Q \mid \mathsf{push}\, n.P \mid \nu n\, P \mid X \mid \mathsf{rec}\, X.P$$

If we restrict P to be the empty process in the production $P ::= n[\,P\,]$ of the grammar above, then this language can be regarded as asynchronous CCS, with the proviso that a process must be enclosed in an *environment* ambient in order to enable pushing of empty ambients to the outside. Consequently, the language is Turing-complete, as it is possible to encode CMs in asynchronous CCS [8]. Theorem 3.5 was proved by encoding asynchronous CCS in L_ν^{op}. We could prove Theorem 3.7 below by similarly encoding asynchronous CCS in L_{rp}, but it is more convenient to encode L_ν^{op} in L_{rp}.

Theorem 3.7 *L_{rp} is Turing-complete.*

PROOF. (Sketch) Let *inert contexts* be defined as

$$\mathcal{C} ::= n[\,\bullet\,] \mid \mathcal{C} \mid n[\,] \mid \nu n\, \mathcal{C}\,.$$

We define $L_{\mathsf{n[rp]}}$ as the set of terms $\mathcal{C}\{P\}$ where P is a term of L_{rp} *with all ambients empty* and $\mathcal{C}\{\bullet\}$ is an inert context. The purpose of $\mathcal{C}\{\bullet\}$ is to make sure that the push operations in P can be executed, by placing P inside an enclosing ambient. Apart from this consideration, contexts cannot perform any reduction at all. We have that $L_{\mathsf{n[rp]}}$ is closed under reductions.

Consider the encoding from L_ν^{op} to L_{rp} which is homomorphic on all terms except for $[\![\mathsf{open}\, n.P]\!] = \mathsf{push}\, n.[\![P]\!]$. For all P in L_ν^{op}, we have that:

(1) for all n, if $P \to Q$ then either
  (a) $n[\,[\![P]\!]\,] \to n[\,[\![Q]\!]\,] \mid m[\,]$, for some m; or
  (b) there are $Q' \in L_\nu^{\mathsf{op}}$ and $m \neq n$ such that $Q \equiv \nu m\, Q'$ and $n[\,[\![P]\!]\,] \to \nu m\,(n[\,[\![Q']\!]\,] \mid m[\,])$;

(2) for all inert contexts $\mathcal{C}\{\bullet\}$ and $R \in L_{\mathsf{n[\,rp\,]}}$, if $\mathcal{C}\{[\![P]\!]\} \to R$ then there are $\mathcal{C}'\{\bullet\}$, $Q, Q' \in L^{\mathsf{op}}_{\nu}$ and m such that $P \to Q$, $Q \equiv \nu m\, Q'$ and $R \equiv \mathcal{C}'\{[\![Q']\!]\}$.

Point (1) shows that there is an effective way to simulate a reduction of L^{op}_{ν} in $L_{\mathsf{n[\,rp\,]}}$ (up to losing an outermost restriction, in case (1b)). Point (2) guarantees that every reduction of a term of $L_{\mathsf{n[\,rp\,]}}$ in the image of the encoding corresponds to a reduction of the original term in L^{op}_{ν} (again up to outermost restriction). The outermost restriction around Q' can be disposed of without altering the behaviour of the term because the resulting term is not composed with any other terms. Both (1) and (2) follow from induction on the derivation of $\to$. □

3.4 "Asynchronous" Languages with open

In this subsection we show that there are Turing-complete AC languages even when we don't allow continuations after movement capabilities. We show this both for objective movement (Theorem 3.8) and for subjective movement (Theorem 3.9).

Let $L^{\mathsf{op}}_{\mathsf{ppa}}$ be the following language (a fragment of PAC):

$$P ::= \mathbf{0} \ \mid\ n[\,P\,] \ \mid\ P \mid Q \ \mid\ \mathsf{open}\, n.P \ \mid\ \mathsf{push}\, n.\mathbf{0} \ \mid\ \mathsf{pull}\, n.\mathbf{0} \ \mid\ !\,\mathsf{open}\, n.P$$

Note that push and pull have no continuation. We might refer to this as *asynchronous* movement. Also, replication is only used with open.

Theorem 3.8 *$L^{\mathsf{op}}_{\mathsf{ppa}}$ is Turing-complete.*

PROOF. We describe an encoding of CMs into $L^{\mathsf{op}}_{\mathsf{ppa}}$. A CM will be encoded as a system consisting of processes encoding the registers in parallel with processes for each instruction.

We consider a particular CM called CM, with instructions $I_0, \ldots, I_a$ and registers $R_0, \ldots, R_b$. Let $CM(i : k_0, \ldots, k_b)$ represent CM when it is about to execute instruction I_i and storing k_j in register j ($j \leq b$). Let the (unique) finite or infinite computation of $CM = CM_0$ be $CM_0, CM_1, \ldots, CM_l, \ldots$, where $CM_l = CM(i_l : k_{0l}, \ldots, k_{bl})$.

First we describe the registers. $R_j(k)$ is encoded as $r_j[\,\underline{k}\,]$, where the numeral process $\underline{k}$ is defined by

$$\underline{0} \overset{\mathrm{df}}{=} z[\,] \qquad \underline{k+1} \overset{\mathrm{df}}{=} s[\,\underline{k}\,]$$

Thus registers are distinguished by their outermost ambient.

In describing the encoding of the instructions, we must take into account the fact that the decrement/jump instructions will accumulate garbage each time they are used, as the code for either decrement or jump is left unused. We therefore parametrise our encoding by the index l of the stage we have reached in the computation. Let $dec(i, l)$ (resp. $jump(i, l)$) be the number of decrements (resp. jumps) performed by instruction i during the computation of CM up to, but not including, stage l.

We denote the encoding of instruction I_i at stage l by $[\![I_i]\!]_l$, defined as follows:

$$
\begin{aligned}
[\![i : \mathsf{Inc}(j)]\!]_l \overset{\text{df}}{=}\ & !\,\mathsf{open}\ \mathsf{st}_i.r_j[\,\mathsf{pull}\ r_j \mid \\
& \quad s[\,\mathsf{pull}\ r_j \mid \mathsf{open}\ r_j.\mathsf{st}_{i+1}[\,] \mid \mathsf{push}\ \mathsf{st}_{i+1}\,] \mid \mathsf{push}\ \mathsf{st}_{i+1}\,]
\end{aligned}
$$

$$
\begin{aligned}
[\![i : \mathsf{DecJump}(j, i')]\!]_l \overset{\text{df}}{=}\ & !\,\mathsf{open}\ \mathsf{st}_i.c_i[\,\mathsf{pull}\ r_j \mid \mathsf{open}\ r_j.(S_{ij} \mid Z_{iji'})\,] \mid \\
& !\,\mathsf{open}\ d_i \mid !\,\mathsf{open}\ d'_i \mid (c_i[\,Z_{iji'}\,])^{dec(i,l)} \mid (c_i[\,S_{ij}\,])^{jump(i,l)}
\end{aligned}
$$

$$
\begin{aligned}
S_{ij} &\overset{\text{df}}{=} d_i[\,\mathsf{pull}\ s \mid r_j[\,\mathsf{pull}\ s \mid \mathsf{open}\ s.(e_i[\,] \mid \mathsf{push}\ e_i)\,] \mid \mathsf{push}\ e_i \mid \mathsf{st}_{i+1}[\,]\,] \mid \\
& \quad \mathsf{open}\ e_i.\mathsf{push}\ d_i \\
Z_{iji'} &\overset{\text{df}}{=} \mathsf{open}\ z.(d'_i[\,r_j[\,\underline{0}\,] \mid \mathsf{st}_{i'}[\,]\,] \mid \mathsf{push}\ d'_i)
\end{aligned}
$$

Notice that the continuations of all occurrences of open are finite (the same condition as used in [14] and mentioned in Section 3.2).

We define:

$$
[\![CM(i : k_0, \ldots, k_b)]\!]_l \overset{\text{df}}{=} \mathsf{st}_i[\,] \mid [\![I_0]\!]_l \mid \cdots \mid [\![I_a]\!]_l \mid r_0[\,\underline{k_0}\,] \mid \cdots \mid r_b[\,\underline{k_b}\,]
$$

The encoding of CM is $[\![CM]\!] \overset{\text{df}}{=} [\![CM_0]\!]_0$. The instructions start without any garbage. The encoded CM will go through successive stages $[\![CM_l]\!]_l$. We show that for each non-terminal stage l, $[\![CM_l]\!]_l \Rightarrow [\![CM_{l+1}]\!]_{l+1}$, and that $[\![CM_l]\!]_l$ is guaranteed to reach $[\![CM_{l+1}]\!]_{l+1}$.

An instruction process $[\![I_i]\!]_l$ is triggered by the presence of st_i at the top level; the instruction starts by consuming st_i. The execution of $[\![I_i]\!]_l$ finishes by unleashing the st_i ambient corresponding to the next instruction. Throughout the computation, at most one st_i ambient is present. The encoded machine terminates if and when the ambient st_{a+1} appears at the top level. There are various cases depending on the nature of the instruction I_i.

An instruction process of the form $[\![i : \mathsf{Inc}(j)]\!]_l$ creates a new register $r_j[\,s[\,]\,]$, which already contains the successor ambient needed to perform the increment. The new register pulls the existing r_j into its core, and strips off the outer casing. The instruction then signals completion by pushing out the trigger for the next instruction. Computation is entirely deterministic. We have:

$$\ldots \mathsf{st}_i[\,] \mid [\![i : \mathsf{Inc}(j)]\!]_l \mid r_j[\,\underline{k}\,] \ldots \Rightarrow \ldots \mathsf{st}_{i+1}[\,] \mid [\![i : \mathsf{Inc}(j)]\!]_{l+1} \mid r_j[\,\underline{k+1}\,] \ldots$$

An instruction process of the form $[\![i : \mathsf{DecJump}(j, i')]\!]_l$ creates a new ambient c_i, pulls in register r_j and strips off its outer layer, leaving the numeral. This numeral has outermost ambient either s or z depending on whether the numeral is zero or a successor.

- If the numeral is a successor it is pulled inside ambient d_i and then inside a new register ambient r_j where it is decremented. The ambient d_i, containing the new incremented register along with the trigger st_{i+1}, is then pushed out of c_i, and opened to unleash the trigger. We have:

$$\begin{aligned} &\ldots \mathsf{st}_i[\,] \mid [\![i : \mathsf{DecJump}(j, i')]\!]_l \mid r_j[\,\underline{k+1}\,] \ldots \\ &\qquad \Rightarrow \ldots \mathsf{st}_{i+1}[\,] \mid [\![i : \mathsf{DecJump}(j, i')]\!]_l \mid c_i[\,Z_{iji'}\,] \mid r_j[\,\underline{k}\,] \ldots \\ &\qquad \equiv \ldots \mathsf{st}_{i+1}[\,] \mid [\![i : \mathsf{DecJump}(j, i')]\!]_{l+1} \mid r_j[\,\underline{k}\,] \ldots \end{aligned}$$

  The execution of the decrement leaves $c_i[\,Z_{iji'}\,]$ behind as garbage, which does not take any further part in the computation. Again, computation is entirely deterministic.
- If the numeral is zero, this is detected by $\mathsf{open}\ z$, and a new ambient d_i, containing $r_j[\,\underline{0}\,]$ along with the trigger $\mathsf{st}_{i'}$, is then pushed out of c_i, and opened to unleash the trigger. We have:

$$\begin{aligned} &\ldots \mathsf{st}_i[\,] \mid [\![i : \mathsf{DecJump}(j, i')]\!]_l \mid r_j[\,\underline{0}\,] \ldots \\ &\qquad \Rightarrow \ldots \mathsf{st}_{i'}[\,] \mid [\![i : \mathsf{DecJump}(j, i')]\!]_l \mid c_i[\,S_{ij}\,] \mid r_j[\,\underline{0}\,] \ldots \\ &\qquad \equiv \ldots \mathsf{st}_{i'}[\,] \mid [\![i : \mathsf{DecJump}(j, i')]\!]_{l+1} \mid r_j[\,\underline{0}\,] \ldots \end{aligned}$$

  Again, computation is entirely deterministic.

Finally, we see that if CM_L is terminal (so that $i_L = a + 1$) then $[\![CM_L]\!]_L$ has no reductions. $[\![CM_L]\!]_L$ displays barb st_{a+1} to indicate termination. The result of the computation, stored in register 0, is usable by subsequent computations. On the other hand, if CM does not terminate, then neither does $[\![CM]\!]$, and the barb st_{a+1} will never appear. There are no “bad” computations, i.e. ones which halt in a non-final state, diverge, or produce unintended behaviour. We have a encoding which shows Turing completeness, and also undecidability of termination and of weak barbs. □

We can achieve exactly the same asynchrony for subjective movement, though the encoding is more elaborate. Let $L^{\mathsf{op}}_{\mathsf{ioa}}$ be the following language:

$$P ::= \mathbf{0} \mid n[\,P\,] \mid P \mid Q \mid \mathsf{open}\, n.P \mid \mathsf{in}\, n.\mathbf{0} \mid \mathsf{out}\, n.\mathbf{0} \mid !\,\mathsf{open}\, n.P$$

Theorem 3.9 *$L^{\mathsf{op}}_{\mathsf{ioa}}$ is Turing-complete. (Proof: see Appendix A.)*

This result improves Theorem 3.6. Moreover, just as with Theorem 3.8, CMs are encoded in such a way that the continuations of all occurrences of open are finite.

3.5 Languages without open

In this subsection we encode CMs into a language with just the standard movement capabilities, namely in and out.

Let L_{io} be the following language:

$$P ::= \mathbf{0} \mid n[\,P\,] \mid P \mid Q \mid \mathsf{in}\, n.P \mid \mathsf{out}\, n.P \mid !\,\mathsf{in}\, n.P \mid !\,\mathsf{out}\, n.P$$

Clearly L_{io} is a sublanguage of $L^{\mathsf{op}}_{\mathsf{io}}$ as defined earlier. The major difference is that L_{io} does not have the open capability. Also, replication is only applied to the capabilities. We shall see in Sections 4 and 5 that the computational strength of a language can depend on whether replication is applied to capabilities or to ambients.

Theorem 3.10 *L_{io} is Turing-complete.*

PROOF. We sketch the encoding of CMs in L_{io} here; see Appendix B for the details. One problem we encountered was in dealing with instructions. Since each instruction I_i has to be used indefinitely many times, one might encode it as $!\,p_i[\,P_i\,]$, where each time the instruction is needed a new copy of $p_i[\,P_i\,]$ is spun off. But then the previously used copies may interfere with the current copy, so that for instance acknowledgements may get misdirected to old p_i ambients still present. This issue would not arise if we could destroy unwanted ambients using the open capability.

Registers consist of a series of double skins $s[\,t[\ldots]\,]$ with $z[\,]$ at the core. We use a double skin rather than the more obvious $s[\,s[\,z[\,]\,]\,]$ style. This is to help with decrementing, which is done by stripping off the outermost s and then in a separate operation stripping off the t ambient now exposed.

We follow Busi and Zavattaro in carrying out the increment of a register by adding a new $s[\,t[\,]\,]$ immediately surrounding the central core $z[\,]$. This seems preferable to adding a new double skin on the outside, since it keeps the increment code and decrement code from interfering with each other.

The basic idea is that each instruction I_i is triggered by entering a st_i ambient. All the other instructions and all the registers enter as well—a monitor process checks that this has happened before I_i is allowed to execute. So the computation goes down a level every time an instruction is executed. When an instruction finishes, it unleashes the st_i ambient to trigger the next instruction. If and when the computation finishes, the first register is sent up to the top level, where it can serve as input for possible further computations.

Therefore we have Turing completeness. Our encoding furthermore establishes that the weak barb relation is undecidable, and that having a non-terminating computation is undecidable.

As the computation proceeds, inert garbage accumulates in both the instructions and the registers. We handle this much as in the proof of Theorem 3.8, letting the encodings of the instructions and the registers be parametrised with the current step in the computation.

The computation is largely deterministic; the exceptions are that, between executions of instructions, the instructions and registers make their way down a level in an indeterminate order, and there is also some limited concurrency in the increment. □

Remark 3.11 *We shall prove that if we remove* out *from* L_{io} *the resulting language is terminating (Theorem 4.8), and similarly if we remove* in *the resulting language is terminating (Theorem 4.14). Since terminating languages cannot be Turing-complete, this will establish that* L_{io} *is a minimal Turing-complete language.*

Remark 3.12 *In independent work, Boneva and Talbot [2] have encoded two-counter machines into the following language:*

$$P ::= \mathbf{0} \;\mid\; n[\,P\,] \;\mid\; P \mid Q \;\mid\; \mathsf{in}\, n.P \;\mid\; \mathsf{out}\, n.P \;\mid\; !\,P$$

(Notice that this language differs slightly from L_{io}*, in that it allows replication of arbitrary processes, including ambients.) However, their encoding can diverge and take wrong turnings into error states, which means that they do not claim Turing completeness. Nevertheless because they establish one-step preservation, they can show that it is undecidable whether one process is reachable from another, and also whether* $P \Downarrow n$ *for an arbitrary process* P *and name* n*.*

It is an open question whether reachability for arbitrary processes in L_{io} is decidable. Even if reachability were decidable for L_{io}, this would not contradict Turing completeness (see Section 3.1).

We have just encoded CMs into language L_{io} with the standard subjective movement capabilities (and without open). We can also encode CMs in the following language L_{pp} with objective moves:

$$P ::= \mathbf{0} \mid n[\,P\,] \mid P \mid Q \mid \mathsf{push}\, n.P \mid \mathsf{pull}\, n.P \mid !\,\mathsf{push}\, n.P \mid !\,\mathsf{pull}\, n.P$$

Theorem 3.13 *L_{pp} is Turing-complete. (Proof: see Appendix C.)*

Remark 3.14 *We shall prove that if we remove* push *from L_{pp} the resulting language is terminating (Theorem 4.8), and if we remove* pull *then termination is decidable for the resulting language (Theorem 5.21).*

4 Terminating Fragments of AC

We would like to know whether the language L_{io} of Section 3.5 is a minimal Turing-complete language. As a partial answer to this question, we shall show in this section that if we remove one of the movement capabilities (either in or out) then the resulting language is in fact terminating, i.e. every computation terminates.

Definition 4.1 *A language $(L, \longrightarrow)$ is* terminating *if every computation is finite.*

In our proofs in this section we shall use a well-founded ordering on multisets. A *multiset* over a set A is a function $S : A \to \mathbb{N}$, where $S(a)$ represents the multiplicity of a in S. A multiset is *finite* if $S(i) = 0$ for all but finitely many $i \in \mathbb{N}$. Let $\mathrm{FMS}(A)$ denote the finite multisets over A.

Definition 4.2 *Suppose that A is partially ordered by $<$. We define $\succ$ to be the transitive closure of the relation between multisets over A where one multiset is obtained from another by replacing an element by any finite number (including zero) of smaller elements.*

An ordering is *well-founded* if it has no infinite decreasing chain.

Proposition 4.3 ([11]) *If $(A, <)$ is a well-founded partial ordering, then so is $(\mathrm{FMS}(A), \prec)$.*

We shall apply this proposition with A as the natural numbers $\mathbb{N}$ with the standard ordering.

4.1 Termination with in

Let $L_{\mathsf{i\bar{i}p}}$ be the following language:

$$\begin{aligned} P ::= \mathbf{0} \ \mid\ n[\,P\,] \ \mid\ P \mid Q \ \mid\ \mathsf{in}\, n.P \ \mid\ \overline{\mathsf{in}}\, n.P \ \mid\ \mathsf{pull}\, n.P \\ \mid\ !\,\mathsf{in}\, n.P \ \mid\ !\,\overline{\mathsf{in}}\, n.P \ \mid\ !\,\mathsf{pull}\, n.P \ \mid\ \nu n\ P \end{aligned}$$

Notice that $L_{\mathsf{i\bar{i}p}}$ is got from L_{io} by removing the out capability and (in order to sharpen the next theorem) adding the co-capability $\overline{\mathsf{in}}$ of SA, the pull of PAC, and restriction. We shall prove that $L_{\mathsf{i\bar{i}p}}$ is terminating (Theorem 4.8 below).

We start by eliminating restriction and pull. Let $m \in \mathcal{N}$ be a single designated name. Let $L^{m}_{\mathsf{i\bar{i}}}$ be the following language:

$$P ::= \mathbf{0} \ \mid\ m[\,P\,] \ \mid\ P \mid Q \ \mid\ \mathsf{in}\, m.P \ \mid\ \overline{\mathsf{in}}\, m.P \ \mid\ !\,\mathsf{in}\, m.P \ \mid\ !\,\overline{\mathsf{in}}\, m.P$$

We define an encoding $[\![-]\!]$ from $L_{\mathsf{i\bar{i}p}}$ to $L^{m}_{\mathsf{i\bar{i}}}$ as follows:

$$\begin{aligned}
[\![\mathbf{0}]\!] &\overset{\mathrm{df}}{=} \mathbf{0} & [\![\mathsf{pull}\, n.P]\!] &\overset{\mathrm{df}}{=} \overline{\mathsf{in}}\, m.[\![P]\!] \\
[\![n[\,P\,]]\!] &\overset{\mathrm{df}}{=} m[\ !\,\mathsf{in}\, m \mid [\![P]\!]\,] & [\![\,!\,\mathsf{in}\, n.P]\!] &\overset{\mathrm{df}}{=} \ !\,\mathsf{in}\, m.[\![P]\!] \\
[\![P \mid Q]\!] &\overset{\mathrm{df}}{=} [\![P]\!] \mid [\![Q]\!] & [\![\,!\,\overline{\mathsf{in}}\, n.P]\!] &\overset{\mathrm{df}}{=} \ !\,\overline{\mathsf{in}}\, m.[\![P]\!] \\
[\![\mathsf{in}\, n.P]\!] &\overset{\mathrm{df}}{=} \mathsf{in}\, m.[\![P]\!] & [\![\,!\,\mathsf{pull}\, n.P]\!] &\overset{\mathrm{df}}{=} \ !\,\overline{\mathsf{in}}\, m.[\![P]\!] \\
[\![\overline{\mathsf{in}}\, n.P]\!] &\overset{\mathrm{df}}{=} \overline{\mathsf{in}}\, m.[\![P]\!] & [\![\nu n\ P]\!] &\overset{\mathrm{df}}{=} [\![P]\!]
\end{aligned}$$

The idea of the encoding is that if we eliminate all restrictions then all existing reductions can still occur (as well as potentially some new ones). Also, making all names the same can only increase the possibility of reductions. Finally, since $L^{m}_{\mathsf{i\bar{i}}}$ has only one name, we can simulate pull by $\overline{\mathsf{in}}$, provided we equip each ambient with $!\,\mathsf{in}\, m$; this again cannot remove any potential reductions, and may well add new ones.

Lemma 4.4 *Let $P, Q \in L_{\mathsf{i\bar{i}p}}$.*

(1) If $P \equiv Q$ then $[\![P]\!] \equiv [\![Q]\!]$.
(2) If $P \to Q$ then $[\![P]\!] \to [\![Q]\!]$.

PROOF. Straightforward and omitted. □

It follows that in order to show that $L_{\overline{\mathsf{ii}}\mathsf{p}}$ is terminating, it is enough to show that $L_{\overline{\mathsf{ii}}}^{m}$ is terminating.

We first define the *capability nesting depth (cnd)* of an $L_{\overline{\mathsf{ii}}}^{m}$ process:

$$
\begin{array}{ll}
\mathsf{cnd}(\mathbf{0}) \overset{\mathrm{df}}{=} 0 & \mathsf{cnd}(\overline{\mathsf{in}}\, m.P) \overset{\mathrm{df}}{=} \mathsf{cnd}(P) + 1 \\
\mathsf{cnd}(m[\,P\,]) \overset{\mathrm{df}}{=} \mathsf{cnd}(P) & \mathsf{cnd}(\,!\,\mathsf{in}\, m.P) \overset{\mathrm{df}}{=} \mathsf{cnd}(P) + 1 \\
\mathsf{cnd}(P \mid Q) \overset{\mathrm{df}}{=} \max(\mathsf{cnd}(P), \mathsf{cnd}(Q)) & \mathsf{cnd}(\,!\,\overline{\mathsf{in}}\, m.P) \overset{\mathrm{df}}{=} \mathsf{cnd}(P) + 1 \\
\mathsf{cnd}(\mathsf{in}\, m.P) \overset{\mathrm{df}}{=} \mathsf{cnd}(P) + 1 &
\end{array}
$$

Note that if $P \equiv Q$ then $\mathsf{cnd}(P) = \mathsf{cnd}(Q)$.

We next define the *capability degree* (abbreviated to cd, or simply degree) of an ambient $m[\,P\,]$. This is the cnd of the *capability component* of P, defined as follows. Any process P is structurally congruent to $P^{\mathsf{cap}} \mid P^{\mathsf{amb}}$, where the capability component P^{cap} is the parallel composition of processes prefixed by capabilities or replicated capabilities, and the ambient component P^{amb} is the parallel composition of ambients. An empty parallel composition is of course the nil process. We let $\mathsf{cd}(m[\,P\,]) \overset{\mathrm{df}}{=} \mathsf{cnd}(P^{\mathsf{cap}})$. This is well-defined with respect to structural congruence. Notice that the degree of an ambient can reduce during a computation, as a result of it entering another ambient. It can never increase. We shall refer to the *initial* degree of an ambient, which is its degree when it first becomes unguarded during a computation. Note also that the degree of an ambient is unaffected by other ambients entering of whatever degree.

During a computation an ambient can produce "children" inside itself, as it enters other ambients. For instance, $m[\,!\,\mathsf{in}\, m.m[\,]\,]$ can produce a series of new $m[\,]$ ambients. These children will have strictly lower capability degrees. For a given ambient $m[\,P\,]$ there is a fixed finite bound on the number of children which can be produced by a single reduction.

Strictly speaking, keeping track of an ambient during a computation relies on labelling ambients. This can be done straightforwardly; we avoid mentioning it further, in order to improve readability.

Proposition 4.5 *$L_{\overline{\mathsf{ii}}}^{m}$ is terminating.*

PROOF. We give two proofs of termination: the first relies on assuming a minimal infinite computation and then showing that there must be a smaller one, while in the second proof we restrict attention to a "top-level" reduction strategy, assign multisets to the processes in a computation and show that they are decreasing in a particular well-founded ordering.

Method 1. Suppose that $P_0 \to \cdots \to P_i \to \ldots$ is an infinite computation. Let D_0 be the maximum of the degrees of the unguarded ambients in P_0. During the computation new ambients are created as children of existing ambients. They will all have initial degree less than their parents, and thus $< D_0$. Since the computation is infinite, infinitely many children must be created (with finitely many ambients computations must be finite, since no pair of ambients can enter each other more than once). Let $D < D_0$ be the maximum degree at which infinitely many children are created. In the whole computation there are only finitely many ambients with initial degree $> D$. At least one of these must be *infinitely productive*, that is, produce infinitely many children. Now let $c > 0$ be the number of infinitely productive ambients of initial degree $> D$.

We have shown how to assign a *value* (D, c) ($D \geq 0$, $c \geq 1$) to each infinite computation. Now let $\mathsf{C} : P_0 \to \cdots$ be an infinite computation with a minimal value of (D, c) in the well-founded lexicographic ordering

$$(D, c) < (D', c') \text{ iff } D < D' \text{ or } (D = D' \text{ and } c < c') \ .$$

We shall obtain a contradiction by showing that there is another infinite computation with a smaller value.

Choose any infinitely productive ambient of initial degree $> D$. We can assume that it is available at the start of C, by removing a finite initial segment of C if necessary (this might reduce D_0, but does not change D and c). Each process P_i of C is of the form $\mathcal{C}\{m[\, C \mid A\,]\}$, where we display the outer context and inner contents of our chosen ambient, with C the capability component, and A the ambient component. There are four types of reduction:

(1) An outer reduction involving the context alone produces $\mathcal{C}'\{m[\, C \mid A\,]\}$.
(2) An inner reduction involving the contents alone produces $\mathcal{C}'\{m[\, C \mid B\,]\}$, where $A \to B$.
(3) The chosen ambient can enter an ambient in the context, producing children A' and resulting in $\mathcal{C}'\{m[\, C' \mid A \mid A'\,]\}$.
(4) The chosen ambient can be entered by an ambient $m[\, R\,]$ in the context, producing children A' and resulting in $\mathcal{C}'\{m[\, C' \mid A \mid A' \mid m[\, R\,]\,]\}$.

Since the ambient is infinitely productive, there must be infinitely many reductions of types (3) or (4).

We shall alter C in two ways. First we remove all type (2) reductions. This does not affect any of the other reductions, since type (1) reductions are independent of the ambient contents, and type (3) or (4) reductions only depend on the capability component C, which is unaffected by type (2) reductions. We get a new computation $\mathsf{C}' : P'_0 \to \cdots \to P'_i \to \cdots$, with $P'_0 = P_0$. It must be infinite, since it still has all the type (3) or (4) reductions of C. Let the value of C' be (D', c'). Any ambients in C' must have already been in C. Hence $(D', c') \leq (D, c)$.

Now let us alter C' by making the chosen ambient totally unproductive, as follows: Suppose that $P_0 = P'_0 = \mathcal{C}_0\{m[\,C_0 \mid A_0\,]\}$. We translate C_0 to C'_0 by replacing any ambient $m[\,R\,]$ by the nil process (and translating all other operators homomorphically). All the reductions of C' can still proceed, since type (1), (3) or (4) reductions do not depend on the ambient component of the chosen ambient, and the same capabilities are exercised by the chosen ambient, even though no children are produced. We get a new infinite computation $\mathsf{C}'' : P''_0 \to \cdots \to P''_i \to \cdots$.

Let the value of C'' be (D'', c''). Any ambients in C'' must have already been in C'. Hence $(D'', c'') \leq (D', c')$. Also we have made an infinitely productive ambient of degree $> D \geq D'$ into one which is totally unproductive. We may or may not have reduced the degree of the chosen ambient, but this does not matter. We have certainly reduced the number of infinitely productive ambients of degree $> D'$. So either $D'' < D'$ or $c'' < c'$. Hence $(D'', c'') < (D', c') \leq (D, c)$. This contradicts the minimality of C. □

Before giving the second method we need some further definitions and lemmas.

Any reduction $P \to Q$ is either "top-level" (i.e. one top-level ambient enters another), or else "lower-level" (the reduction occurs inside a top-level ambient). In formal terms, the difference is that rule (Amb) (Section 2) is used in the latter case but not in the former. Let us write $P \to_{top} Q$ for a top-level reduction and $P \to_{lower} Q$ for a lower-level reduction. The reflexive and transitive closures are denoted by $\Rightarrow_{top}$, $\Rightarrow_{lower}$ respectively.

Lemma 4.6 *Let P, Q be L^m_{ii} processes. If $P \Rightarrow_{lower}\to_{top} Q$ then $P \to_{top}\Rightarrow_{lower} Q$.*

PROOF. Straightforward and omitted. □

Let us write $P \searrow Q$ if $P \equiv m[\,Q\,] \mid R$ for some R.

Lemma 4.7 *Let P be a L^m_{ii} process. Suppose that P has an infinite computation $P \to^\omega$. Then $P \Rightarrow_{top}\searrow\to^\omega$.*

PROOF. The computation $P \to^{\omega}$ will have finitely many $\to_{top}$ reductions. Using Lemma 4.6 we can transform it into another infinite computation with all $\to_{top}$ reductions carried out at the beginning: $P \Rightarrow_{top} P' \to^{\omega}_{lower}$. Then P' must have at least one top-level ambient, and there must be an infinite computation inside one of these top-level ambients $m[\,Q\,]$. So $P \Rightarrow_{top} P' \searrow Q \to^{\omega}$ as required. □

Proof of Proposition 4.5, Method 2. Let P be an $L^{m}_{\overline{\mathsf{ii}}}$ process. From Lemma 4.7, we see that if P has an infinite $\to$ computation then P has an infinite $\Rightarrow_{top}\searrow$ computation. To show that infinite $\Rightarrow_{top}\searrow$ computations are impossible, we assign multisets to processes and define an ordering on these multisets which is well-founded and strictly decreasing with respect to $\Rightarrow_{top}\searrow$.

For a completely formal proof we would have to develop an apparatus for labelling ambients and members of multisets in order to make precise the correspondence between the two. We have suppressed all of this in the interests of readability.

Let $P_0, \ldots, P_i, \ldots$ be an infinite $\Rightarrow_{top}\searrow$ computation (i.e. $P_i \to_{top} P_{i+1}$ or $P_i \searrow P_{i+1}$, and there are infinitely many i for which $P_i \searrow P_{i+1}$). We assign to each P_i a finite multiset S_i. Its elements will be ordered pairs (d, T) consisting of a natural number d and a finite multiset T of natural numbers. The multiset S_i will satisfy the following:

(1) For each $(d, T) \in S_i$, and for each $d' \in T$ we have $d' < d$.
(2) The numbers in S_i are precisely all degrees of unguarded ambients in P_i: there is a bijective correspondence which maps each unguarded ambient $m[\,Q\,]$ of P_i to $d \geq \mathsf{cd}(m[\,Q\,])$ in S_i, either as the left-hand component of some $(d, T) \in S_i$ or as some $d \in T$ where $(d', T) \in S_i$.
(3) If $m[\,Q\,]$ occurs at the top level in P_i, then $m[\,Q\,]$ corresponds to d in some (d, T) in S_i.
(4) If $m[\,R\,]$ corresponds to $d' \in T$ for some (d, T) in S_i, then $m[\,R\,]$ is unguarded inside some $m[\,Q\,]$ corresponding to d.

We create S_0 as follows: For each unguarded ambient $m[\,P'\,]$ of degree d contained in P_0, we add the pair $(d, \emptyset)$ to S_0. Plainly properties (1–4) are established.

In the computation there are two kinds of reductions: $\to_{top}$ and $\searrow$. Suppose that $P_i \to_{top} P_{i+1}$. A $\to_{top}$ reduction consists of an ambient $m[\,Q_1\,]$ of degree d_1 entering an ambient $m[\,Q_2\,]$ of degree d_2. To these ambients there correspond elements (d'_1, T_1) and (d'_2, T_2) in S_i, with $d_1 \leq d'_1$ and $d_2 \leq d'_2$. (Since we are doing a top-level reduction the two ambients are represented in the first elements of the pairs of S_i, by (3).) The $\to_{top}$ reduction will produce children of $m[\,Q_1\,]$ of degree $< d_1$; we add their degrees to T_1. The reduction will also produce children of $m[\,Q_2\,]$ of degree $< d_2$; we add their degrees to T_2. In this way we create S_{i+1}. It is easy to check that properties (1–4) are established for S_{i+1}.

Now suppose that $P_i \searrow P_{i+1}$. The $\searrow$ reduction selects a top-level ambient $m[\,P_{i+1}\,]$, and keeps P_{i+1} while discarding its enclosing ambient and any other top-level processes in parallel with $m[\,P_{i+1}\,]$. Suppose that $m[\,P_{i+1}\,]$ is of degree d_0 and corresponds to the element (d'_0, T_0) of S_i. First we remove from S_i all pairs corresponding to the discarded top-level processes and their contents. Note that by (3) and (4), if any member of some (d, T) is to be removed, then so are all the remaining members. Now we remove (d'_0, T_0) from S_i, and for each $d \in T_0$ we add $(d, \emptyset)$ to S_i. Note that each $d < d_0 \leq d'_0$. In this way we create S_{i+1}.

Properties (1), (2) and (4) are clearly established for S_{i+1}. As to (3), suppose that $m[\,R\,]$ is a top-level ambient in P_{i+1}. Suppose that $m[\,R\,]$ corresponds to $d' \in T$ for some (d, T) in S_{i+1}. Then this (d, T) was already in S_i. Therefore by (4) for S_i, $m[\,R\,]$ was inside some $m[\,Q\,]$ corresponding to d. The only way that $m[\,R\,]$ can be top-level in P_{i+1} is for $m[\,Q\,]$ to be $m[\,P_{i+1}\,]$, which means that $m[\,R\,]$ corresponds to d' in some $(d', \emptyset)$ in S_{i+1}. Thus we have established (3).

Recall the well-founded ordering on multisets of Definition 4.2 and Proposition 4.3. If we consider just the first members of the pairs in the multisets S_i we see that a $\rightarrow_{top}$ reduction leaves the set unchanged, while a $\searrow$ reduction removes one element and replaces it with a finite set of smaller elements (it also removes zero or more elements completely, corresponding to the discarded top-level processes). So each $\Rightarrow_{top}\searrow$ reduction takes us down in the $\succ$ ordering. By well-foundedness of $\succ$ there is no infinite $\Rightarrow_{top}\searrow$ computation, and thus no infinite $\rightarrow$ computation. □

Theorem 4.8 *$L_{\mathsf{i\bar{i}p}}$ is terminating.*

PROOF. By Lemma 4.4 and Proposition 4.5. □

4.2 Termination with out

It is also the case that a language with out as its only capability is terminating. Let L_{o} be the following language:

$$P ::= \mathbf{0} \mid n[\,P\,] \mid P \mid Q \mid \mathsf{out}\, n.P \mid !\,\mathsf{out}\, n.P \mid \nu n\, P$$

Notice that L_{o} is got from L_{io} (Section 3.5) by removing the in capability and (in order to sharpen the next theorem) adding restriction. We shall show that L_{o} is terminating (Theorem 4.14 below).

The strategy we adopt is as follows: Firstly, as with Theorem 4.8, it suffices to show that the sublanguage without restriction, and where all names are identified, is terminating. We associate a finite multiset of natural numbers with each process and show that each reduction produces a smaller multiset in the well-founded ordering $\succ$ of Definition 4.2. As the multiset is sensitive to the number of nil processes and unfoldings of replications, we have to use a non-standard notion of reduction.

We start by eliminating restriction. Let $m \in \mathcal{N}$ be a single designated name. Let L_{o}^{m} be the following language:

$$P ::= \mathbf{0} \mid m[\,P\,] \mid P \mid Q \mid \mathsf{out}\, m.P \mid \,!\,\mathsf{out}\, m.P$$

We define an encoding $[\![-]\!]$ from L_{o} to L_{o}^{m} as follows:

$$\begin{array}{rlrl}
[\![\mathbf{0}]\!] & \stackrel{\text{df}}{=} \mathbf{0} & [\![\mathsf{out}\, n.P]\!] & \stackrel{\text{df}}{=} \mathsf{out}\, m.[\![P]\!] \\
[\![n[\,P\,]]\!] & \stackrel{\text{df}}{=} m[\,[\![P]\!]\,] & [\![\,!\,\mathsf{out}\, n.P]\!] & \stackrel{\text{df}}{=} \,!\,\mathsf{out}\, m.[\![P]\!] \\
[\![P \mid Q]\!] & \stackrel{\text{df}}{=} [\![P]\!] \mid [\![Q]\!] & [\![\nu n\, P]\!] & \stackrel{\text{df}}{=} [\![P]\!]
\end{array}$$

Lemma 4.9 *Let $P, Q \in L_{\mathsf{o}}$.*

(1) If $P \equiv Q$ then $[\![P]\!] \equiv [\![Q]\!]$.
(2) If $P \rightarrow Q$ then $[\![P]\!] \rightarrow [\![Q]\!]$.

PROOF. Straightforward and omitted. □

We associate a finite multiset of natural numbers with each process of L_{o}^{m}. Each element in the multiset measures the number of ambients working from an occurrence of $\mathbf{0}$ outwards.

$$\begin{array}{ll}
\mathsf{ms}(\mathbf{0}) \stackrel{\text{df}}{=} \{0\} & \mathsf{ms}(\mathsf{out}\, m.P) \stackrel{\text{df}}{=} \mathsf{ms}(P) \\
\mathsf{ms}(m[\,P\,]) \stackrel{\text{df}}{=} \{k+1 : k \in \mathsf{ms}(P)\} & \mathsf{ms}(\,!\,\mathsf{out}\, m.P) \stackrel{\text{df}}{=} \mathsf{ms}(P) \\
\mathsf{ms}(P \mid Q) \stackrel{\text{df}}{=} \mathsf{ms}(P) \cup \mathsf{ms}(Q) &
\end{array}$$

Notice that this definition will produce different multisets for processes which are structurally congruent. For instance, $\mathsf{ms}(m[\,\mathbf{0}\,]) = \{1\}$, while $\mathsf{ms}(m[\,\mathbf{0} \mid$

$\mathbf{0}\,]) = \{1, 1\}$. Also, $\mathsf{ms}(\,!\,\mathsf{out}\,m.\mathbf{0}) = \{0\}$, while $\mathsf{ms}(\mathsf{out}\,m.\mathbf{0} \mid !\,\mathsf{out}\,m.\mathbf{0}) = \{0, 0\}$. We therefore replace $\equiv$ by commutative-associative structural congruence $\equiv_{ca}$ (Section 2), where the rules $\mathbf{0} \mid P \equiv P$ and $P \equiv P \mid !\,P$ are disallowed.

Having adjusted structural congruence, we also need to change to a non-standard reduction relation $\rightarrow'$. We replace the usual rule (Out) by the following:

$$\text{(Out1)} \qquad m[\,m[\,\mathsf{out}\,m.P \mid Q\,] \mid R\,] \rightarrow' m[\,P \mid Q\,] \mid m[\,R\,]$$

$$\text{(RepOut)}\ m[\,m[\,!\,\mathsf{out}\,m.P \mid Q\,] \mid R\,] \rightarrow' m[\,P \mid !\,\mathsf{out}\,m.P \mid Q\,] \mid m[\,R\,]$$

The rule (RepOut) ensures that replication is only unfolded as needed. Since we no longer can add nil processes using structural congruence, the two new rules also come with variants where Q is not present, and where R is not present (and we write $\mathbf{0}$ instead of R in the derivative). The remaining rules are:

$$\frac{P \rightarrow' P'}{n[\,P\,] \rightarrow' n[\,P'\,]} \qquad \frac{P \rightarrow' P'}{P \mid Q \rightarrow' P' \mid Q} \qquad \frac{P \equiv_{ca} P' \quad P' \rightarrow' Q' \quad Q' \equiv_{ca} Q}{P \rightarrow' Q}$$

We next show that we have not removed any possibilities for computation by changing the reduction relation.

Lemma 4.10 *Let P, Q be L^m_{o} processes.*

(1) If $P \equiv\rightarrow' Q$ then $P \rightarrow'\equiv Q$.
(2) If $P \rightarrow Q$ then $P \rightarrow'\equiv Q$.

PROOF. Lengthy and omitted. It is similar to [25, Lemma 1.4.15] and [8, Propositions 4.11, 4.12], but in those cases a labelled transition system was being related to an unlabelled one, whereas here we are relating two unlabelled transition systems. □

Lemma 4.11 *Let P be an L^m_{o} process. If P has an infinite $\rightarrow$ computation, then P has an infinite $\rightarrow'$ computation.*

PROOF. Suppose that there is an infinite computation

$$P = P_0 \rightarrow \cdots \rightarrow P_i \rightarrow \cdots .$$

We create an infinite computation $P = P'_0 \rightarrow' \cdots \rightarrow' P'_i \rightarrow' \cdots$, with $P_i \equiv P'_i$ (all i), defining P'_i by induction as follows. Suppose that $P = P'_0 \rightarrow' \cdots \rightarrow' P'_i$

with $P'_i \equiv P_i$. We have $P_i \rightarrow P_{i+1}$. Hence by Lemma 4.10(2) there is Q such that $P_i \rightarrow' Q \equiv P_{i+1}$. Therefore $P'_i \equiv P_i \rightarrow' Q$. By Lemma 4.10(1) there is P'_{i+1} such that $P'_i \rightarrow' P'_{i+1} \equiv Q$. Clearly $P'_{i+1} \equiv P_{i+1}$, and P'_{i+1} is as required. □

Now we establish that $\rightarrow'$ reductions take us down in the $\prec$ multiset ordering of Definition 4.2.

Lemma 4.12 *Let P, Q be L^m_{o} processes, and let $\mathcal{C}\{\bullet\}$ be an L^m_{o} context.*

(1) If $\mathsf{ms}(P) = \mathsf{ms}(Q)$ then $\mathsf{ms}(\mathcal{C}\{P\}) = \mathsf{ms}(\mathcal{C}\{Q\})$.
(2) If $\mathsf{ms}(P) \succ \mathsf{ms}(Q)$ then $\mathsf{ms}(\mathcal{C}\{P\}) \succ \mathsf{ms}(\mathcal{C}\{Q\})$.
(3) If $P \equiv_{ca} Q$ then $\mathsf{ms}(P) = \mathsf{ms}(Q)$.
(4) If $P \rightarrow' Q$ then $\mathsf{ms}(P) \succ \mathsf{ms}(Q)$.

PROOF. (1) and (2) are by structural induction on contexts. (3) uses (1), and is straightforward. (4) uses (2) and (3), and is by induction on the derivation of $P \rightarrow' Q$. As an example, consider the rule (RepOut) in the case where Q and R are omitted:

$$m[\, m[\, !\,\mathsf{out}\, m.P\,]\,] \rightarrow' m[\, P \mid !\,\mathsf{out}\, m.P\,] \mid m[\,\mathbf{0}\,]$$

We have

$$\mathsf{ms}(m[\, m[\, !\,\mathsf{out}\, m.P\,]\,]) = \{k+2 : k \in \mathsf{ms}(P)\}$$
$$\mathsf{ms}(m[\, P \mid !\,\mathsf{out}\, m.P\,] \mid m[\,\mathbf{0}\,]) = \{k+1 : k \in \mathsf{ms}(P) \cup \mathsf{ms}(P)\} \cup \{1\}$$

Clearly $\{k+2 : k \in \mathsf{ms}(P)\} \succ \{k+1 : k \in \mathsf{ms}(P) \cup \mathsf{ms}(P)\} \cup \{1\}$. We omit further details. □

Proposition 4.13 *(1) $(L^m_{\mathsf{o}}, \rightarrow')$ is terminating.*
(2) $(L^m_{\mathsf{o}}, \rightarrow)$ is terminating.

PROOF.

(1) This follows from Lemma 4.12 and the well-foundedness of $\succ$ (Proposition 4.3).
(2) This follows from (1) and Lemma 4.11.

□

Our main result now follows:

Theorem 4.14 *$L_{\circ}$ is terminating.*

PROOF. This follows from Proposition 4.13(2) and Lemma 4.9(2). □

Remark 4.15 *From the proof of Lemma 4.12(4) we see that a single reduction $P \rightarrow' Q$ leads to at most 3 smaller items being substituted for each element ≥ 2 of $\mathsf{ms}(P)$. An example is*

$$m[\,m[\,!\,\mathsf{out}\,m.\mathbf{0}\,]\,] \rightarrow' m[\,\mathbf{0} \mid !\,\mathsf{out}\,m.\mathbf{0}\,] \mid m[\,\mathbf{0}\,]\,,$$

where $\{2\}$ becomes $\{1,1,1\}$. Thus if an $L_{\circ}^{m}$ process has $\leq k$ $\mathbf{0}$s and $\leq k$ ambients, its multiset will be bounded by $\{k,\ldots,k\}$ (k copies of k), and the maximum length of a computation will be bounded by $k.3^{k-1}$. This upper bound also applies to $(L_{\circ}^{m}, \rightarrow)$ computations by the proof of Lemma 4.11, and to $(L_{\circ}, \rightarrow)$ computations by Lemma 4.9. We obtain that if an $L_{\circ}$ process has $\leq k$ operators then any computation has length bounded by $k.3^{k-1}$. This bound can no doubt be considerably improved.

Notice that we *can* have infinite computations in the language where we add co-capability $\overline{\mathsf{out}}$ to $L_{\circ}$, in view of the counterexample

$$n[\,n[\,\mathsf{out}\,n\,] \mid !\,\overline{\mathsf{out}}\,n.n[\,\mathsf{out}\,n\,]\,]\,.$$

This is equally the case when the co-capability is located at the upper level [17]:

$$n[\,n[\,\mathsf{out}\,n\,]\,] \mid !\,\overline{\mathsf{out}}\,n.n[\,n[\,\mathsf{out}\,n\,]\,]$$

With “push” as the only capability we can have infinite computations, e.g.

$$n[\,n[\;] \mid !\,\mathsf{push}\,n.n[\;]\,]\,.$$

Remark 4.16 *If we combine replication with the* open *capability we can create non-terminating processes such as $n[\;] \mid !\,\mathsf{open}\,n.n[\;]$. Busi and Zavattaro [8] showed that termination is decidable for processes built with replication and* open *(see Theorem 5.21 in Section 5).*

5 Fragments of AC with Decidable Termination

We have seen (Theorem 3.10) that pure Boxed Ambients is Turing-complete. In the previous section we saw that the fragments with just one movement capability (either in or out), and replication just applied to that capability, are terminating. In this section we look at the same fragments, but extended with full replication. We shall show that termination is decidable in the fragments with in (respectively out) and full replication. In the case of out, we shall be able to go further and show that the fragment with out, open and full (unboxed) recursion has decidable termination. In the next subsection we start with this result, which builds on the work of Busi and Zavattaro.

Definition 5.1 *We shall say that* termination is decidable *in a language* $(L, \rightarrow)$ *if, given any process* P *of* L*, it is decidable whether* P *has an infinite computation.*

Remark 5.2 *We saw in Section 3.1 that having decidable termination does not* per se *imply that a language is Turing-incomplete. Nevertheless, whenever we showed that a language was Turing-complete, it was also the case that termination was undecidable. This enables us to achieve a separation between such languages and the ones discussed in this section. See Remark 5.22 below.*

5.1 Decidability for out *and* open

Recall that Busi and Zavattaro [8] showed that pure MA, with no movement capabilities and with (unboxed) recursion rather than replication, is Turing-complete (Theorem 3.5). They also showed that if one replaces recursion by what they call *unrestricted recursion* then termination is decidable. (Recursion is said to be unrestricted if, for each process $\mathsf{rec}\, X.P$, no free occurrence of X in P occurs inside a subprocess of the form $\nu n\, Q$.) Their language, which we shall call $L^{\mathsf{op}}_{\nu,ur}$, is defined by

$$P ::= \mathbf{0} \mid n[\,P\,] \mid P \,|\, Q \mid \mathsf{open}\, n.P \mid \nu n\, P \mid X \mid \mathsf{rec}\, X.P$$

where recursion is unboxed and unrestricted.

Theorem 5.3 ([8]) *Termination is decidable for* $L^{\mathsf{op}}_{\nu,ur}$*.*

In particular, termination is decidable both in the sublanguage with open, restriction and replication, and in the sublanguage with open and recursion (but not restriction).

The proof of Theorem 5.3 depends on the theory of well-quasi-orderings and well-structured transition systems [12]. We briefly review the relevant definitions and results.

Definition 5.4 *A* quasi-ordering *(qo) is a reflexive and transitive binary relation. A* well-quasi-ordering *(wqo) is a qo* $(X, \leq)$ *such that for every infinite* *sequence $x_0, \dots, x_i, \dots$ of members of X, there exist $i, j \in \mathbb{N}$ such that $i < j$ and $x_i \leq x_j$.*

Definition 5.5 *A* transition system $(S, \rightarrow)$ *is a set of states S together with a transition relation $\rightarrow$. For $s \in S$ let $\mathsf{Succ}(s) \overset{\text{df}}{=} \{t : s \rightarrow t\}$ and let $\mathsf{Deriv}(s) \overset{\text{df}}{=} \{t : s \Rightarrow t\}$. $(S, \rightarrow)$ is* finite-branching *if for all $s \in S$, $\mathsf{Succ}(s)$ is finite.*

Definition 5.6 *A structure $(S, \rightarrow, \leq)$ is a* well-structured transition system (with strong compatibility) *if*

- *$(S, \rightarrow)$ is a transition system, and*
- *$\leq$ is a wqo on S, and*
- *$\leq$ is upwards compatible with $\rightarrow$, meaning that if $s \rightarrow t$ and $s \leq s'$ then there exists t' such that $s' \rightarrow t'$ and $t \leq t'$.*

Theorem 5.7 *(Special case of [12, Theorem 4.6]) Let $(S, \rightarrow, \leq)$ be a well-structured transition system (with strong compatibility) where $\leq$ is decidable and $\mathsf{Succ}(s)$ is finite and computable in s. Then it is decidable, given $s \in S$, whether there is an infinite $\rightarrow$ computation starting from s.*

In order to apply this theorem to $L^{\mathsf{op}}_{\nu,ur}$, Busi and Zavattaro firstly need to show that $\mathsf{Succ}(P)$ is computable. The problem is that the standard reduction relation is not finite-branching, since it allows recursions to be unfolded without limit (using structural congruence). They therefore define a different reduction relation using a labelled transition system, which only allows unfolding as required to perform a reduction.

Next they define a multiset-style ordering $\preceq$ on processes, under which, for example, $P \preceq P \mid Q$. In showing that $\preceq$ is a wqo, the essential ingredients are:

(1) Bounded depth: there is a bound on the depth of all derivatives of a process (in terms of nesting of ambients and restrictions), and
(2) Finite name-space: the set of names used in all derivatives of a process is finite.

The bounded depth property comes straightforwardly from the facts that recursion is unboxed and that there are no movement capabilities. The finite name-space property comes from the fact that recursion is unrestricted, so that it is never necessary to extrude the scope of a restriction, with the renaming that this entails.

We wish to extend Busi and Zavattaro's work by applying it to a fragment with the out capability. The starting point is to note that an out reduction can never increase depth. Therefore we can fulfil the bounded depth property. In order to fulfil the finite name-space property we find it necessary to disallow restriction. The reason is that with ambient movement it becomes essential to extrude scopes, and so we may need to create unboundedly many new names during a computation to avoid clashes. This is true even if we reduce the language by replacing unrestricted recursion with replication (any replication can be seen as an unrestricted recursion, but not, of course, vice versa). Consider for example

$$\begin{aligned}
&m[\,!\,\nu n\,(n[\,\mathsf{out}\,m\,] \mid n[\,]\,)\,] \\
&\to \nu n_1\,(n_1[\,] \mid m[\,n_1[\,] \mid !\,\nu n\,(n[\,\mathsf{out}\,m\,] \mid n[\,]\,)\,]) \\
&\to \nu n_1 n_2\,(n_1[\,] \mid n_2[\,] \mid m[\,n_1[\,] \mid n_2[\,] \mid !\,\nu n\,(n[\,\mathsf{out}\,m\,] \mid n[\,]\,)\,]) \\
&\ldots
\end{aligned}$$

There is no way to make the scopes of $n_1, n_2, \ldots$ disjoint, and so the computation uses infinitely many names.

We shall show that termination is decidable for the following language, which we call $L_{\mathsf{o}}^{\mathsf{op}}$:

$$\begin{aligned}
P ::= \mathbf{0} \mid n[\,P\,] \mid P \mid Q \mid \mathsf{open}\,n.P \mid \overline{\mathsf{open}}\,n.P \\
\mid \mathsf{out}\,n.P \mid \overline{\mathsf{out}}\,n.P \mid \mathsf{push}\,n.P \mid X \mid \mathsf{rec}\,X.P
\end{aligned}$$

(Recursion is unboxed in $L_{\mathsf{o}}^{\mathsf{op}}$.) The rest of this subsection is devoted to proving this result (Theorem 5.21 below).

First we need to change from standard reduction to one which is finite-branching. With ambient movement it is problematic to use labelled transition systems (as did Busi and Zavattaro). Therefore we go directly to a finite-branching notion of reduction. We shall define what we call *unfolding reduction*, which means that we unfold each recursion exactly once for each reduction.

Looking at the rules for → given in Section 2, we see that the infinite branching comes from the following two rules of structural congruence:

$$\mathbf{0} \mid P \equiv P \qquad \mathsf{rec}\,X.P \equiv P\{\mathsf{rec}\,X.P/X\}$$

The first allows indefinitely many nil processes to accumulate, while the second allows us to unfold recursions indefinitely many times, even for a single reduction. We therefore remove these rules from structural congruence, and use commutative-associative structural congruence $\equiv_{ca}$ (Section 2). Notice that this is exactly what we used in Section 4.2 when proving that L_{o} is terminating. In fact, the non-standard notion of reduction we defined there is finitely branching, though we did not require that for the proof.

Next we define a non-standard notion of reduction $\rightarrow_{ca}$ for $L_{\mathsf{o}}^{\mathsf{op}}$. This has the same rules as normal reduction, with two changes:

(1) Much as when we defined a non-standard reduction in Section 4.2, we include variants of the rules (SafeOpen), (SafeOut) and (Push) which allow for the possible absence of processes in parallel with capabilities. This is unnecessary with standard reduction where the law $\mathbf{0} \mid P \equiv P$ is available.

(2) We replace $\equiv$ by $\equiv_{ca}$ in rule (Str):

$$\frac{P \equiv_{ca} P' \quad P' \rightarrow_{ca} Q' \quad Q' \equiv_{ca} Q}{P \rightarrow_{ca} Q}$$

Since we have removed the rule of structural congruence which allows unfolding of recursions, before performing a reduction we unfold each recursion exactly once, producing what we call the *unfolding* of a process.

Definition 5.8 *The (single)* unfolding $\mathsf{unf}(P)$ *of an* $L_{\mathsf{o}}^{\mathsf{op}}$ *term* P *is defined as follows:*

$$\mathsf{unf}(\mathsf{rec}\, X.P) \overset{\mathrm{df}}{=} \mathsf{unf}(P)\{\mathsf{rec}\, X.P/X\}\,,$$

with $\mathsf{unf}(P)$ *being defined homomorphically for all other operators of* $L_{\mathsf{o}}^{\mathsf{op}}$*.*

As an example, let

$$P \overset{\mathrm{df}}{=} \mathsf{out}\, n.(X \mid \mathsf{rec}\, Y.Q) \qquad Q \overset{\mathrm{df}}{=} X \mid Y \mid n[\,]$$

Then $\mathsf{unf}(\mathsf{rec}\, X.P) = \mathsf{out}\, n.(\mathsf{rec}\, X.P \mid (\mathsf{rec}\, X.P \mid \mathsf{rec}\, Y.Q \mid n[\,]))$.

The unfolding of a process allows every possible immediate reduction to go ahead. The fact that a single unfolding is enough depends on the particular operators of $L_{\mathsf{o}}^{\mathsf{op}}$. If we were dealing with in, then for instance $\mathsf{rec}X.(X \mid m[\mathsf{in}\, m\,])$ needs to be unfolded twice to expose the redex. The difference between in and out is that an in-redex involves two operators of the same kind (namely, ambients) at the same depth. Although an out-redex involves two ambients, they are at different levels (one being inside the other). Since recursion is unboxed, if an out-redex is exposed by a second unfolding, it (or an essentially identical redex) must have been exposed by the first unfolding.

Definition 5.9 *(Unfolding Reduction)* $P \rightarrow_u Q$ *iff* $\mathsf{unf}(P) \rightarrow_{ca} Q$*.*

We must show that $\rightarrow_u$ is finite-branching, and that a process has an infinite $\rightarrow$-computation iff it has an infinite $\rightarrow_u$-computation. First we prove some lemmas. It is convenient to split structural congruence $\equiv$ into two component notions:

Definition 5.10 *(1) Let $\equiv_{nca}$ be the least congruence on $L_{\mathsf{o}}^{\mathsf{op}}$ terms generated by the following laws:*

$$\mathbf{0} \mid P \equiv_{nca} P \qquad P \mid Q \equiv_{nca} Q \mid P \qquad (P \mid Q) \mid R \equiv_{nca} P \mid (Q \mid R)$$

(2) Let $\rhd$ be the least precongruence on $L_{\mathsf{o}}^{\mathsf{op}}$ terms generated by

$$\mathsf{rec}\, X.P \rhd P\{\mathsf{rec}\, X.P/X\}$$

Thus $\equiv_{nca}$ is $\equiv_{ca}$ with the law for the nil process added. We get $\rhd$ by treating the law $\mathsf{rec}\, X.P \equiv P\{\mathsf{rec}\, X.P/X\}$ as a rewrite rule. Any derivation of $P \equiv Q$ is a chain of instances of $\equiv_{nca}$ and $\rhd$ and its inverse $\lhd$.

Lemma 5.11 *Let P, Q, R, S be $L_{\mathsf{o}}^{\mathsf{op}}$ terms and X a process variable.*

(1) If $P \equiv_{nca} Q$ and $R \equiv_{nca} S$ then $P\{R/X\} \equiv_{nca} Q\{S/X\}$.
(2) If $P \rhd Q$ and $R \rhd S$ then $P\{R/X\} \rhd Q\{S/X\}$.

PROOF. Straightforward and omitted. □

Lemma 5.12 *Let P be an $L_{\mathsf{o}}^{\mathsf{op}}$ term. Then $P \rhd \mathsf{unf}(P)$.*

PROOF. By structural induction on terms. All cases are immediate except for recursion. So suppose that $P \rhd \mathsf{unf}(P)$. We must show that $\mathsf{rec}\, X.P \rhd \mathsf{unf}(\mathsf{rec}\, X.P)$. Now $\mathsf{rec}\, X.P \rhd P\{\mathsf{rec}\, X.P/X\}$ and

$$\mathsf{unf}(\mathsf{rec}\, X.P) = \mathsf{unf}(P)\{\mathsf{rec}\, X.P/X\}\,.$$

So the result follows from Lemma 5.11. □

Lemma 5.13 *Let P, Q be $L_{\mathsf{o}}^{\mathsf{op}}$ terms.*

(1) If $P \equiv_{nca} Q$ then $\mathsf{unf}(P) \equiv_{nca} \mathsf{unf}(Q)$.
(2) If $P \rhd Q$ then $\mathsf{unf}(P) \rhd \mathsf{unf}(Q)$.

PROOF.

(1) Structural induction on terms for each of the three laws of $\equiv_{nca}$, using Lemma 5.11.

(2) By induction on the derivation of $P \rhd Q$. The only case which is not immediate is recursion. Suppose that $P = \mathsf{rec}\, X.P'$ and $P \rhd Q$ is got by a single unfolding. There are two possibilities for Q. Either Q is got by unfolding the outermost recursion, or it is got by unfolding some recursion inside P'. In the first case we have $Q = P'\{P/X\}$. Then $\mathsf{unf}(P) = \mathsf{unf}(P')\{P/X\}$ and $\mathsf{unf}(Q) = \mathsf{unf}(P')\{\mathsf{unf}(P)/X\}$. So $\mathsf{unf}(P) \rhd \mathsf{unf}(Q)$ by Lemma 5.11. In the second case we have $Q = \mathsf{rec}\, X.Q'$ with $P' \rhd Q'$ in one step. Then $\mathsf{unf}(Q) = \mathsf{unf}(Q')\{Q/X\}$. By inductive hypothesis, $\mathsf{unf}(P') \rhd \mathsf{unf}(Q')$, and $\mathsf{unf}(P) \rhd \mathsf{unf}(Q)$ by Lemma 5.11.

□

Lemma 5.14 *Let P, Q be $L_{\mathsf{o}}^{\mathsf{op}}$ processes.*

(1) If $P \equiv_{nca} \rightarrow_{ca} Q$ then $P \rightarrow_{ca} \equiv_{nca} Q$.
(2) If $P \lhd \rightarrow_{ca} Q$ then $P \rightarrow_{ca} \lhd Q$.
(3) If $P \rhd \rightarrow_{u} Q$ then $P \rightarrow_{u} \equiv Q$.
(4) If $P \equiv \rightarrow_{u} Q$ then $P \rightarrow_{u} \equiv Q$.
(5) $P \rightarrow Q$ iff $P \rightarrow_{u} \equiv Q$.

PROOF.

(1) Straightforward and omitted.

(2) Suppose that $P \lhd P'$ in a single step. Then there is a context $\mathcal{C}\{\bullet\}$ such that $P' = \mathcal{C}\{\mathsf{rec}\, X.R\}$ and $P = \mathcal{C}\{R\{\mathsf{rec}\, X.R/X\}\}$. Suppose $P' \rightarrow_{ca} Q$. Then $Q = \mathcal{C}'\{\mathsf{rec}\, X.R\}$ for some $\mathcal{C}'\{\bullet\}$. Furthermore $P \rightarrow_{ca} Q'$ where $Q' = \mathcal{C}'\{R\{\mathsf{rec}\, X.R/X\}\}$. Clearly $Q' \lhd Q$ as required.

(3) Notice that the converse of (2) does not hold: it is not the case that if $P \rhd \rightarrow_{ca} Q$ then $P \rightarrow_{ca} \rhd Q$, since in general unfolding recursions can create new redexes.

The idea is that if a recursion has already been unfolded once, unfolding a second time does not give any new redexes.

Suppose that $P \rhd P'$ in a single step and $\mathsf{unf}(P') \rightarrow_{ca} Q$. Then there is a context $\mathcal{C}\{\bullet\}$ such that $P = \mathcal{C}\{R\}$, with $R = \mathsf{rec}\, X.R_1$, and $P' = \mathcal{C}\{R'\}$, with $R' = R_1\{R/X\}$.

Now $\mathsf{unf}(\mathcal{C}\{R\}) = \sigma_R(\mathcal{C}'\{\mathsf{unf}(R)\})$ for some context $\mathcal{C}'\{\bullet\}$, where σ_R assigns recursive terms to any variables bound by recursion in $\mathcal{C}\{\bullet\}$. Also $\mathsf{unf}(\mathcal{C}\{R'\}) = \sigma_{R'}(\mathcal{C}'\{\mathsf{unf}(R')\})$, where we have $\sigma_R(Y) \rhd \sigma_{R'}(Y)$ for any variables bound by recursion in $\mathcal{C}\{\bullet\}$.

As a simple example, let $\mathcal{C}\{\bullet\} \overset{\mathrm{df}}{=} \mathsf{rec}\, Y.(Y \mid \bullet)$. Then $\mathsf{unf}(\mathcal{C}\{R\}) = \sigma_R(Y \mid \mathsf{unf}(R))$, where $\sigma_R(Y) = \mathcal{C}\{R\}$, and $\mathsf{unf}(\mathcal{C}\{R'\}) = \sigma_{R'}(Y \mid \mathsf{unf}(R'))$, where $\sigma_{R'}(Y) = \mathcal{C}\{R'\}$.

We have $\mathsf{unf}(R_1) \equiv_{ca} X^i \mid R_{\mathsf{a}} \mid R_{\mathsf{c}} \mid R_{\mathsf{r}} \mid R_{\mathsf{n}}$, where $i \geq 0$, R_{a} is the parallel composition of ambient terms, R_{c} is the parallel composition of capability terms of the form $M.P$, R_{r} is the parallel composition of recursive terms, and R_{n} is the parallel composition of nil processes. To be precise, any or all of R_{a} , R_{c} , R_{r} , R_{n} may be absent from $\mathsf{unf}(R_1)$. Note that X does not occur free in R_{a} , since recursion is unboxed. Hence:

$$\begin{aligned}
\mathsf{unf}(R) =\ & \mathsf{unf}(R_1)\{R/X\} \\
\equiv_{ca}\ & R^i \mid R_{\mathsf{a}} \mid R_{\mathsf{c}}\{R/X\} \mid R_{\mathsf{r}}\{R/X\} \mid R_{\mathsf{n}} \\
\mathsf{unf}(R') =\ & \mathsf{unf}(R_1\{R/X\}) \\
=\ & \mathsf{unf}(R_1)\{\mathsf{unf}(R)/X\} \\
\equiv_{ca}\ & (\mathsf{unf}(R))^i \mid R_{\mathsf{a}} \mid R_{\mathsf{c}}\{\mathsf{unf}(R)/X\} \mid R_{\mathsf{r}}\{\mathsf{unf}(R)/X\} \mid R_{\mathsf{n}}
\end{aligned}$$

Notice that in $\mathsf{unf}(R')$ there are now $i+1$ copies of R_{a}. Also there are i copies of $R_{\mathsf{c}}\{R/X\}$.

Now we look at how the reduction $\mathsf{unf}(P') = \sigma_{R'}(\mathcal{C}'\{\mathsf{unf}(R')\}) \rightarrow_{ca} Q$ can arise. Inspection of the redexes for open, out and push shows that within $\mathsf{unf}(R')$ at most one ambient and one capability can be involved. Also the only possible movement is of an ambient term, which does not involve $\mathsf{unf}(R)$. The subtlety is that the capability involved may come from either $R_{\mathsf{c}}\{R/X\}$ or $R_{\mathsf{c}}\{\mathsf{unf}(R)/X\}$. Working up to $\equiv$, we can assume that the ambient term used is the rightmost one. Also, if the capability term used is a $R_{\mathsf{c}}\{R/X\}$, we can replace it by $R_{\mathsf{c}}\{\mathsf{unf}(R)/X\}$ and get a result equivalent under $\equiv$, using the fact that $R \equiv \mathsf{unf}(R)$ by Lemma 5.12. Hence

$$\mathsf{unf}(P') \rightarrow_{ca} \sigma_{R'}(\mathcal{C}''\{\mathsf{unf}(R)\}) \equiv Q\,,$$

for some context $\mathcal{C}''\{\bullet\}$. This reduction can be mimicked by

$$\mathsf{unf}(P) \rightarrow_{ca} \sigma_{R}(\mathcal{C}''\{R\})\,.$$

Now $\sigma_{R'}(\mathcal{C}''\{\mathsf{unf}(R)\}) \equiv \sigma_R(\mathcal{C}''\{R\})$. Hence $P \rightarrow_u\equiv Q$ as required.

(4) Follows from (1), (2) and (3), using Lemma 5.13.

(5) ($\Rightarrow$) By induction on the derivation of $P \rightarrow Q$, using (4).
($\Leftarrow$) Suppose $P \rightarrow_u\equiv Q$. Then by Lemma 5.12 and the definition of $\rightarrow_u$, $P \rhd \rightarrow_{ca}\equiv Q$. Hence $P \rightarrow Q$.

□

Lemma 5.15 *Let P be an $L_{\mathsf{o}}^{\mathsf{op}}$ process. Then P has an infinite $\rightarrow$-computation iff P has an infinite $\rightarrow_u$-computation.*

PROOF. ($\Rightarrow$) Much the same as Lemma 4.11, using Lemma 5.14.
($\Leftarrow$) If there is an infinite $\rightarrow_u$-computation $P \rightarrow_u P_1 \rightarrow_u \cdots$, then $P \rightarrow P_1 \rightarrow$

$\cdots$ is an infinite $\rightarrow$-computation, by Lemma 5.14(5). $\square$

Lemma 5.16 *(1) For any $L_{\circ}^{\mathsf{op}}$ term P, $\{Q : P \equiv_{ca} Q\}$ is finite.*
(2) $\rightarrow_u$ is finite-branching, and $\mathsf{Succ}(P)$ is computable in P.

PROOF. By structural induction on processes. Omitted. $\square$

Now we follow Busi and Zavattaro and define an ordering $\preceq$ on processes, which will be a wqo:

Definition 5.17 *(cf. [8, Definition 4.17]) Let P, Q be $L_{\circ}^{\mathsf{op}}$ processes. Let $P \preceq Q$ iff*

(1) $Q \equiv_{ca} P \mid R$ for some R, or
(2) $P \equiv_{ca} P_1 \mid n[\,P_2\,]$ and $Q \equiv_{ca} Q_1 \mid n[\,Q_2\,]$, with $P_1 \preceq Q_1$ and $P_2 \preceq Q_2$.

Definition 5.18 *The* ambient nesting depth *of an $L_{\circ}^{\mathsf{op}}$ term is defined as follows:*

$$\begin{array}{rlrl}
\mathsf{and}(\mathbf{0}) & \stackrel{\text{df}}{=} 0 & \mathsf{and}(M.P) & \stackrel{\text{df}}{=} \mathsf{and}(P) \\
\mathsf{and}(n[\,P\,]) & \stackrel{\text{df}}{=} 1 + \mathsf{and}(P) & \mathsf{and}(X) & \stackrel{\text{df}}{=} 0 \\
\mathsf{and}(P \mid Q) & \stackrel{\text{df}}{=} \max(\mathsf{and}(P), \mathsf{and}(Q)) & \mathsf{and}(\mathsf{rec}\, X.P) & \stackrel{\text{df}}{=} \mathsf{and}(P)
\end{array}$$

Reductions do not increase depth:

Lemma 5.19 *(1) Let P, Q be $L_{\circ}^{\mathsf{op}}$ terms. If $P \equiv Q$ then $\mathsf{and}(P) = \mathsf{and}(Q)$.*
(2) Let P, Q be $L_{\circ}^{\mathsf{op}}$ processes. If $P \rightarrow_u Q$ then $\mathsf{and}(P) \geq \mathsf{and}(Q)$.

PROOF. Straightforward and omitted. Note that the proof depends on recursion being unboxed. $\square$

Proposition 5.20 *Let P be an $L_{\circ}^{\mathsf{op}}$ process. Then $(\mathsf{Deriv}(P), \rightarrow_u, \preceq)$ is a well-structured transition system with decidable $\preceq$ and computable $\mathsf{Succ}(-)$.*

PROOF. (Sketch) We show that $\preceq$ is a decidable wqo on $\mathsf{Deriv}(P)$ (using Lemma 5.19, which gives us the Bounded Depth property), and that it is upwards compatible with $\rightarrow_u$. We omit the details, referring the reader to the proof of [8, Theorem 4.29]. We know from Lemma 5.16 that $\mathsf{Succ}(-)$ is computable. $\square$

We can now prove the main theorem of this subsection:

Theorem 5.21 *Termination is decidable for $L_{\mathsf{o}}^{\mathsf{op}}$.*

PROOF. Termination of $\rightarrow_u$-computations is decidable for $L_{\mathsf{o}}^{\mathsf{op}}$ by Theorem 5.7 and Proposition 5.20. Therefore by Lemma 5.15 termination of $\rightarrow$-computations is decidable for $L_{\mathsf{o}}^{\mathsf{op}}$. □

Remark 5.22 *We know that termination is undecidable for L_{io} (see proof of Theorem 3.10). It follows from Theorem 5.21 that there can be no embedding $[\![-]\!]$ from L_{io} into $L_{\mathsf{o}}^{\mathsf{op}}$ which respects termination, in the sense that for any process P of L_{io}, P has a non-terminating computation iff $[\![P]\!]$ has a non-terminating computation.*

5.2 Decidability for in

We now turn to showing that termination is decidable for a language with the in capability and full replication (rather than replication on capabilities, as considered in Section 4). We start by noting that even such a simple process as $!\,n[\,\mathsf{in}\,n\,]$ can have a computation with unbounded ambient nesting depth. The proof method of Theorem 5.21 is therefore not available.

Let L_{in} be the following language:

$$P ::= \mathbf{0} \;\mid\; n[\,P\,] \;\mid\; P \mid Q \;\mid\; \mathsf{in}\,n.P \;\mid\; !\,P$$

We shall show that termination is decidable for L_{in} (Theorem 5.34 below). Our strategy is first to remove all replications except those on capabilities and ambients. Next we define a non-standard notion of reduction which detects any possible divergence and terminates the computation immediately.

Let L'_{in} be the following language:

$$P ::= \mathbf{0} \;\mid\; n[\,P\,] \;\mid\; P \mid Q \;\mid\; \mathsf{in}\,n.P \;\mid\; !\,n[\,P\,] \;\mid\; !\,\mathsf{in}\,n.P$$

We see that L'_{in} is the sublanguage of L_{in} got by requiring that replication can only be applied to ambients and in. Note that if P is a process of L'_{in} and $P \rightarrow Q$ then Q is also a process of L'_{in}.

Define an encoding $[\![-]\!]$ from L_{in} to L'_{in} homomorphically except for replication, where we let

$$[\![\,!\,\mathbf{0}]\!] \stackrel{\mathrm{df}}{=} \mathbf{0} \qquad [\![\,!\,(P \mid Q)]\!] \stackrel{\mathrm{df}}{=} \,!\,[\![P]\!] \mid !\,[\![Q]\!] \qquad [\![\,!\,!\,P]\!] \stackrel{\mathrm{df}}{=} \,!\,[\![P]\!]$$

$$[\![\,!\,n[\,P\,]]\!] \stackrel{\mathrm{df}}{=} \,!\,n[\,[\![P]\!]\,] \qquad [\![\,!\,\mathsf{in}\,n.P]\!] \stackrel{\mathrm{df}}{=} \,!\,\mathsf{in}\,n.[\![P]\!]$$

We next define a non-standard notion of structural congruence $\equiv^!$ on L_{in}. It is the least congruence generated by the usual laws of standard structural congruence appropriate for the operators of L_{in} (Section 2), together with the following:

$$!\,\mathbf{0} \equiv^! \mathbf{0} \qquad !\,!\,P \equiv^! \,!\,P \qquad !\,(P \mid Q) \equiv^! \,!\,P \mid !\,Q$$

These laws are to be found in for instance [14].

Lemma 5.23 *For any L_{in} process P, $P \equiv^! [\![P]\!]$.*

PROOF. By structural induction on L_{in} processes. Omitted. □

Lemma 5.24 *For any L_{in} processes P, Q, if $P \equiv^! \rightarrow Q$ then $P \rightarrow \equiv^! Q$.*

PROOF. By induction on the derivation of $\equiv^!$. Omitted. □

Lemma 5.25 *Let P be an L_{in} process. Then P has an infinite computation iff $[\![P]\!]$ has an infinite computation.*

PROOF. Follows immediately from Lemma 5.23 and Lemma 5.24. □

To decide whether a process P of L'_{in} has a non-terminating computation, we shall define a non-standard reduction relation $\rightarrow^D$ which is finite-branching and which traps non-termination finitely, so that every computation terminates.

As in Section 5.1, in order to achieve finite branching we use commutative-associative structural congruence $\equiv_{ca}$ instead of standard structural congruence $\equiv$. Since we omit the rules for nil and replication we compensate with extra reduction rules, much as in Section 4.2. These ensure that replications are unfolded once as needed:

$$
\begin{array}{ll}
\text{(In1)} & n[\,\mathsf{in}\, m.P \mid Q\,] \mid m[\,R\,] \rightarrow^D m[\,n[\,P \mid Q\,] \mid R\,] \\
\text{(In2)} & n[\,!\,\mathsf{in}\, m.P \mid Q\,] \mid m[\,R\,] \rightarrow^D m[\,n[\,P \mid !\,\mathsf{in}\, m.P \mid Q\,] \mid R\,] \\
\text{(In3)} & n[\,\mathsf{in}\, m.P \mid Q\,] \mid !\,m[\,R\,] \rightarrow^D m[\,n[\,P \mid Q\,] \mid R\,] \mid !\,m[\,R\,] \\
\text{(In4)} & n[\,!\,\mathsf{in}\, m.P \mid Q\,] \mid !\,m[\,R\,] \rightarrow^D m[\,n[\,P \mid !\,\mathsf{in}\, m.P \mid Q\,] \mid R\,] \mid !\,m[\,R\,]
\end{array}
$$

Also, because we cannot add in nil using structural congruence to match a redex, for each of the above four rules there is another rule which is the same except that Q is not composed in parallel. We omit these rules.

$$
\text{(Amb)}\ \frac{P \rightarrow^D P'}{n[\,P\,] \rightarrow^D n[\,P'\,]} \qquad \text{(Str)}\ \frac{P \equiv_{ca} P' \quad P' \rightarrow^D Q' \quad Q' \equiv_{ca} Q}{P \rightarrow^D Q}
$$

$$
\text{(Par)}\ \frac{P \rightarrow^D P'}{P \mid Q \rightarrow^D P' \mid Q}
$$

We introduce a new constant DIV which represents divergence, and which can occur only on the right-hand side of $\rightarrow^D$. Thus $\rightarrow^D \subseteq L'_{\mathsf{in}} \times (L'_{\mathsf{in}} \cup \{\mathsf{DIV}\})$. We add the following rules which trap divergence caused by replicated ambients being able to perform repeated ins:

$$
\begin{array}{ll}
\text{(InDiv1)} & !\,n[\,\mathsf{in}\, m.P \mid Q\,] \mid m[\,R\,] \rightarrow^D \mathsf{DIV} \\
\text{(InDiv2)} & !\,n[\,!\,\mathsf{in}\, m.P \mid Q\,] \mid m[\,R\,] \rightarrow^D \mathsf{DIV} \\
\text{(InDiv3)} & !\,n[\,\mathsf{in}\, m.P \mid Q\,] \mid !\,m[\,R\,] \rightarrow^D \mathsf{DIV} \\
\text{(InDiv4)} & !\,n[\,!\,\mathsf{in}\, m.P \mid Q\,] \mid !\,m[\,R\,] \rightarrow^D \mathsf{DIV} \\
\text{(InDiv5)} & !\,n[\,\mathsf{in}\, n.P \mid Q\,] \rightarrow^D \mathsf{DIV} \\
\text{(InDiv6)} & !\,n[\,!\,\mathsf{in}\, n.P \mid Q\,] \rightarrow^D \mathsf{DIV}
\end{array}
$$

As previously, there are another six rules like the above but with Q missing. Another form of divergence associated with replicated ambients is trapped by the next rule:

$$
\text{(AmbDiv)}\ \frac{P \rightarrow^D P'}{!\,n[\,P\,] \rightarrow^D \mathsf{DIV}}
$$

Finally we add four rules to propagate derivations of DIV:

$$\text{(DivAmb)}\quad \frac{P \to^D \mathsf{DIV}}{n[\,P\,] \to^D \mathsf{DIV}} \qquad \text{(DivRep)}\quad \frac{P \to^D \mathsf{DIV}}{!\,n[\,P\,] \to^D \mathsf{DIV}}$$

$$\text{(DivPar)}\quad \frac{P \to^D \mathsf{DIV}}{P \mid Q \to^D \mathsf{DIV}} \qquad \text{(DivStr)}\quad \frac{P \equiv_{ca} P' \quad P' \to^D \mathsf{DIV}}{P \to^D \mathsf{DIV}}$$

Notice that DIV has no reductions. We complete the definition of $\to^D$ by stipulating that DIV takes priority over any derivative in L'_{in}:

- If $P \to^D \mathsf{DIV}$ then $P \not\to^D Q$ for all $Q \in L'_{\mathsf{in}}$.

We need this condition, because otherwise we could have infinite $\to^D$ computations, which we wish to avoid. An example is $P \overset{\mathrm{df}}{=} m[\,\mathsf{in}\, m\,] \mid !\, m[\,\mathsf{in}\, m\,]$. We have $P \to^D \mathsf{DIV}$, but without the priority condition we would also have the infinite computation $P \to^D m[\,m[\,] \mid \mathsf{in}\, m\,] \mid !\, m[\,\mathsf{in}\, m\,] \to^D \cdots$.

Lemma 5.26 $(L'_{\mathsf{in}}, \to^D)$ *is finite-branching and, given $P \in L'_{\mathsf{in}}$, we can effectively compute its successors under $\to^D$.*

PROOF. Straightforward and omitted. □

The constant DIV represents the finite detection of divergence. We see from the various rules for DIV that the possible causes of divergence are all very simple. What is not so obvious is that the rules have indeed trapped all possible causes of divergence; there might be deeper or more elaborate causes. To rule this out, we shall need to show that $(L'_{\mathsf{in}}, \to^D)$ is terminating, and this is where the main work will lie in proving that termination is decidable for L_{in}.

First we establish the relationship between $\to$ and $\to^D$:

Lemma 5.27 *Let P, Q be L'_{in} processes.*

(1) If $P \to^D Q$ then $P \to Q$.
(2) If $P \to^D \mathsf{DIV}$ then $P \to^\omega$.
(3) If $P \equiv\to^D \mathsf{DIV}$ then $P \to^D \mathsf{DIV}$.
(4) If $P \equiv\to^D Q$ then $P \to^D\equiv Q$.
(5) If $P \to Q$ then either $P \to^D\equiv Q$ or $P \to^D \mathsf{DIV}$.

PROOF.

(1) Straightforward and omitted.

(2) Any reduction $P \to^D \mathsf{DIV}$ must arise from one of the twelve (InDiv) rules, or from (AmbDiv). In each case it is easy to construct an infinite $\to$-computation.

(3) By induction on the derivation of $\equiv$. The idea is that the derivation of DIV is unaffected by whether replications are folded or unfolded, since the various rules for DIV work directly on replicated processes. We omit the details.

(4) By induction on the derivation of $\equiv$. Much as in the previous item, there is no need to unfold replications in order to obtain reductions, since we have rules (In2)-(In4) as well as the standard (In1). We omit the details.

(5) By induction on the derivation of $P \to Q$, using (3) and (4).

□

Having obtained the desired finite-branching transition system $\to^D$, we now complete the proof that termination for $(L'_{\mathsf{in}}, \to)$ is decidable by showing that $(L'_{\mathsf{in}}, \to^D)$ is terminating. We adapt Method 2 for showing that $L^m_{\bar{\mathsf{ii}}}$ is terminating (Section 4.1).

As in Section 4.1, let us write $P \to^D_{top} Q$ for a top-level reduction (one that does not use the rule (Amb)) and $P \to^D_{lower} Q$ for a lower-level reduction (one that does use (Amb)). We do not make this distinction for $P \to^D \mathsf{DIV}$ reductions. The next lemma is similar to Lemma 4.6.

Lemma 5.28 *Let P, Q be L'_{in} processes. If $P \Rightarrow^D_{lower} \to^D_{top} Q$ then $P \to^D_{top} \Rightarrow^D_{lower} Q$.*

PROOF. Straightforward and omitted. □

Lemma 5.29 *$(L'_{\mathsf{in}}, \to^D_{top})$ is terminating.*

PROOF. Take any L'_{in} process P. We have

$$P \equiv_{ca} P^{\mathsf{cap}} \mid \prod_{i \in I} m_i[\, Q_i \,] \mid \prod_{j \in J} !\, n_j[\, R_j \,] \mid \prod_{k \in K} \mathbf{0}\,,$$

where P^{cap} is the capability component of P (see Section 4.1). Any $\to^D_{top}$ computation starting from P will be finite; in fact, it can have no more than $|I|$ reductions. This is because each $m_i[\, Q_i \,]$ can perform at most one top-level in; also at no stage in the computation can we have any top-level reduction where an ambient spun-off from some $!\, n_j[\, R_j \,]$ enters another ambient (as this would imply $P \to^D \mathsf{DIV}$, which would prevent any $\to^D_{top}$ reductions). □

As in Section 4.1, let us write $P \searrow Q$ if $P \equiv m[\,Q\,] \mid R$ for some R. The next lemma is similar to Lemma 4.7.

Lemma 5.30 *Let P be a L'_{in} process. Suppose that P has an infinite computation $P(\rightarrow^D)^\omega$. Then $P \Rightarrow^D_{top} \searrow (\rightarrow^D)^\omega$.*

PROOF. The proof is on the lines of that of Lemma 4.7. Suppose $P(\rightarrow^D)^\omega$. By Lemma 5.29 the computation $P(\rightarrow^D)^\omega$ will have finitely many $\rightarrow^D_{top}$ reductions. Using Lemma 5.28 we can transform $P(\rightarrow^D)^\omega$ into another infinite computation with all $\rightarrow^D_{top}$ reductions carried out at the beginning: $P \Rightarrow^D_{top} P'(\rightarrow^D_{lower})^\omega$. Then P' must have at least one top-level (unreplicated) ambient, and there must be an infinite computation inside one of these top-level ambients $r[\,P''\,]$. So $P \Rightarrow^D_{top} P' \searrow P''(\rightarrow^D)^\omega$ as required. □

Recall that in Section 4.1 we defined the *capability degree* of an ambient. We need to adapt that definition to the present language L'_{in}, where we have replicated ambients. First we define the *capability and replicated ambient depth* of a process:

$$\begin{array}{rlrl}
\mathsf{crad}(\mathbf{0}) & \stackrel{\mathrm{df}}{=} 0 & \mathsf{crad}(\mathsf{in}\ n.P) & \stackrel{\mathrm{df}}{=} \mathsf{crad}(P)+1 \\
\mathsf{crad}(n[\,P\,]) & \stackrel{\mathrm{df}}{=} \mathsf{crad}(P) & \mathsf{crad}(\,!\,n[\,P\,]) & \stackrel{\mathrm{df}}{=} \mathsf{crad}(P)+1 \\
\mathsf{crad}(P \mid Q) & \stackrel{\mathrm{df}}{=} \max(\mathsf{crad}(P),\mathsf{crad}(Q)) & \mathsf{crad}(\,!\,\mathsf{in}\ n.P) & \stackrel{\mathrm{df}}{=} \mathsf{crad}(P)+1
\end{array}$$

Note that this definition increases depth for capabilities and replicated ambients. Next we define the *degree* of an ambient or replicated ambient:

$$\mathsf{degree}(n[\,P\,]) \stackrel{\mathrm{df}}{=} \mathsf{crad}(P^{\mathsf{cap}}) \qquad \mathsf{degree}(\,!\,n[\,P\,]) \stackrel{\mathrm{df}}{=} \mathsf{crad}(\,!\,n[\,P\,])$$

The idea is that the degree of an ambient is unaffected by other ambients entering. Also, if an ambient unleashes "child" ambients or replicated ambients inside itself as a result of entering another ambient, such children will have lower degree. Moreover, if a replicated ambient $!\,n[\,P\,]$ spins off $n[\,P\,]$ then $n[\,P\,]$ and all unguarded ambients and replicated ambients inside $n[\,P\,]$ will have lower degree than $!\,n[\,P\,]$. Note that the degree of an ambient can decrease as a result of that ambient performing an in.

Lemma 5.31 *$(L'_{\mathsf{in}}, \rightarrow^D)$ is terminating.*

PROOF. Let P be an L'_{in} process. From Lemma 5.30, we see that if P has an infinite $\rightarrow^D$ computation then P has an infinite $\Rightarrow^D_{top}\searrow$ computation. To show that infinite $\Rightarrow^{D}_{top}\searrow$ computations are impossible, we assign multisets to processes and define an ordering on these multisets which is well-founded and strictly decreasing with respect to $\Rightarrow^{D}_{top}\searrow$.

As when using Method 2 to show that L^{m}_{ii} is terminating, for a completely formal proof we would have to develop an apparatus for labelling ambients and members of multisets in order to make precise the correspondence between the two. We would also have to keep track of which ambients are spun off from which occurrences of replicated ambients. Again we have suppressed all of this in the interests of readability.

Let $P_0, \ldots, P_i, \ldots$ be an infinite $\Rightarrow^{D}_{top}\searrow$ computation (i.e. $P_i \rightarrow^{D}_{top} P_{i+1}$ or $P_i \searrow P_{i+1}$, and there are infinitely many i for which $P_i \searrow P_{i+1}$). We assign to each P_i a finite multiset S_i. Its elements will be ordered pairs (d, T) consisting of a natural number d and a finite multiset T of natural numbers. Let us say that an ambient (or replicated ambient) is *IR-guarded* if it occurs inside the scope of an in (or $!\,\mathsf{in}$), or inside a replicated ambient. The negation of IR-guarded is *IR-unguarded*. The multiset S_i will satisfy the following:

(1) For each $(d, T) \in S_i$, and for each $d' \in T$ we have $d' < d$.

(2) The numbers in S_i are precisely all degrees of IR-unguarded ambients and IR-unguarded replicated ambients in P_i: there is a bijective correspondence which
  (a) maps each IR-unguarded ambient $m[\,Q\,]$ of P_i to $d \geq \mathsf{degree}(m[\,Q\,])$ in S_i, and
  (b) maps each IR-unguarded replicated ambient $!\,m[\,Q\,]$ of P_i to number $d = \mathsf{degree}(\,!\,m[\,Q\,])$ in S_i,

  either as the left-hand component of some $(d, T) \in S_i$ or as some $d \in T$ where $(d', T) \in S_i$.

(3) (a) If $m[\,Q\,]$ occurs at the top level in P_i, and $m[\,Q\,]$ was not spun off from some top-level replicated ambient, then $m[\,Q\,]$ corresponds to d in some (d, T) in S_i.
  (b) If $!\,m[\,Q\,]$ occurs at the top level in P_i, then $!\,m[\,Q\,]$ corresponds to d in some (d, T) in S_i.

(4) If $m[\,R\,]$ (resp. $!\,m[\,R\,]$) corresponds to $d' \in T$ for some (d, T) in S_i, then $m[\,R\,]$ (resp. $!\,m[\,R\,]$) is IR-unguarded inside some $m[\,Q\,]$ corresponding to d, or else d corresponds to a top-level IR-unguarded replicated ambient.

We create S_0 as follows: For each IR-unguarded ambient $m[\,Q\,]$ of degree d contained in P_0, we add the ordered pair $(d, \emptyset)$ to S_0. Similarly, for each IR-unguarded replicated ambient $!\,m[\,Q\,]$ of degree d contained in P_0, we add the ordered pair $(d, \emptyset)$ to S_0. Plainly properties (1–4) are established.

In the computation there are two kinds of reductions: $\rightarrow^{D}_{top}$ and $\searrow$. Suppose that $P_i \rightarrow^{D}_{top} P_{i+1}$. There are two kinds of $\rightarrow^{D}_{top}$ reduction:

- An ambient $m_1[\,Q_1\,]$ of degree d_1 enters an ambient $m_2[\,Q_2\,]$, using rules (In1) or (In2).
- An ambient $m_1[\,Q_1\,]$ of degree d_1 enters an ambient $m_2[\,Q_2\,]$ spun off from $!\,m_2[\,Q_2\,]$ of degree d_2, using rules (In3) or (In4).

In both cases, by (3), $m_1[\,Q_1\,]$ corresponds to d'_1 in (d'_1, T_1), with $d_1 \leq d'_1$. The reduction may produce children within $m_1[\,Q_1\,]$, and we add their degrees (which are less than d_1) to T_1. In the second case, by (3), $!\,m_2[\,Q_2\,]$ corresponds to d_2 in (d_2, T_2). We have new IR-unguarded ambients and replicated ambients produced by spinning off $m_2[\,Q_2\,]$; we add their degrees (which are less than d_2) to T_2. In this way we create S_{i+1}. It is easy to check that properties (1–4) are established for S_{i+1}.

Now suppose that $P_i \searrow P_{i+1}$. The $\searrow$ reduction selects a top-level ambient $m[\,P_{i+1}\,]$, and keeps P_{i+1} while discarding its enclosing ambient and any other top-level processes in parallel with $m[\,P_{i+1}\,]$. To create S_{i+1}, first we remove each top-level replicated ambient, and remove from S_i the corresponding (d, T), replacing it by $(d', \emptyset)$ for each $d' \in T$. Call this new set S'_i. Suppose that $m[\,P_{i+1}\,]$ is of degree d_0. By (33a) and the construction it corresponds to an element (d'_0, T_0) of S'_i. Secondly we remove all other top-level ambients together with their contents. We remove the corresponding entries in S'_i. Note that by (3), (4) and the construction, if any member of some (d, T) is to be removed, then so are all the remaining members. Thirdly we remove (d'_0, T_0) from S'_i, and for each $d \in T_0$ we add $(d, \emptyset)$ to S'_i. Note that each $d < d_0 \leq d'_0$. Only this third stage is guaranteed to take us down in the multiset ordering. In this way we create S_{i+1}.

Properties (1), (2) and (4) are clearly established for S_{i+1}. As to (3), suppose that $(\,!\,)m[\,R\,]$ is a top-level ambient or replicated ambient in P_{i+1} which corresponds to $d' \in T$ for some (d, T) in S_{i+1}. Then this (d, T) was already in S_i. Therefore by (4) for S_i and the construction of S_{i+1}, $(\,!\,)m[\,R\,]$ was inside some $m[\,Q\,]$ corresponding to d. The only way that $(\,!\,)m[\,R\,]$ can be top-level in P_{i+1} is for $m[\,Q\,]$ to be $m[\,P_{i+1}\,]$, which means that $(\,!\,)m[\,R\,]$ corresponds to d' in some $(d', \emptyset)$ in S_{i+1}. Thus we have established (3).

Recall the well-founded ordering on multisets of Definition 4.2 and Proposition 4.3. If we consider just the first members of the pairs in the multisets S_i, we see that a $\rightarrow^{D}_{top}$ reduction leaves the set unchanged, while a $\searrow$ reduction removes one element and replaces it with a finite set of smaller elements (it also removes zero or more elements completely, corresponding to the discarded top-level processes). So each $\Rightarrow^{D}_{top}\searrow$ reduction takes us down in the $\succ$ ordering. By well-foundedness of $\succ$ there is no infinite $\Rightarrow^{D}_{top}\searrow$ computation, and thus no infinite $\rightarrow^{D}$ computation. $\Box$

Lemma 5.32 *Let P be an L'_{in} process. Then P has an infinite $\rightarrow$-computation iff $P \Rightarrow^D \mathsf{DIV}$.*

PROOF. ($\Rightarrow$) Suppose $P = P_0 \rightarrow \cdots \rightarrow P_i \rightarrow \cdots$ is an infinite computation. Assume for a contradiction that it is not the case that $P \Rightarrow^D \mathsf{DIV}$. We shall construct by induction an infinite $\rightarrow^D$-computation, which contradicts Lemma 5.31. Let $P'_0 = P_0$. Suppose that we have $P'_0 \rightarrow^D \cdots \rightarrow^D P'_i$ with $P'_j \equiv P_j$ for all $j \leq i$. Since $P'_i \equiv P_i \rightarrow P_{i+1}$, we have $P'_i \rightarrow P_{i+1}$, and by Lemma 5.27(5) there exists P'_{i+1} such that $P'_i \rightarrow^D P'_{i+1}$ and $P'_{i+1} \equiv P_{i+1}$, since $P'_i \rightarrow^D \mathsf{DIV}$ is impossible by assumption.
($\Leftarrow$) Suppose $P \Rightarrow^D \mathsf{DIV}$. Then $P \rightarrow^\omega$ by Lemma 5.27(1,2). □

Lemma 5.33 *Termination is decidable for $(L'_{\mathsf{in}}, \rightarrow)$.*

PROOF. To decide whether P has an infinite computation, by Lemma 5.32 we need only check whether $P \Rightarrow^D \mathsf{DIV}$. We do this by computing the entire computation tree of P under $\rightarrow^D$ (of course, we can stop if and when DIV is encountered). This is possible by Lemma 5.26 and Lemma 5.31. □

We can now state our main theorem:

Theorem 5.34 *Termination is decidable for L_{in}.*

PROOF. By Lemma 5.25 and Lemma 5.33. □

Remark 5.35 *It is an open question whether termination is decidable when L_{in} is extended with safe in as in SA. The proof method used for Theorem 5.34 appears not to work, since it relies on defining a non-standard reduction relation which is terminating. The difficulty is to find such a relation for which there can be no infinite top-level computation (as shown for $(L'_{\mathsf{in}}, \rightarrow^D_{top})$ in Lemma 5.29). Here is an example to show the extra complications that arise with SA:*

$$m[\,\mathsf{in}\, m_1\,] \mid !\, m_1[\,\overline{\mathsf{in}}\, m_1.\mathsf{in}\, m_2\,] \mid !\, m_2[\,\overline{\mathsf{in}}\, m_2.\mathsf{in}\, m_3\,] \mid \cdots \mid !\, m_k[\,\overline{\mathsf{in}}\, m_k.\mathsf{in}\, m_1\,]$$

Here $m[\,\mathsf{in}\, m_1\,]$ acts as a catalyst to set in motion a cycle of k top-level reductions, which can repeat without end. This divergence can only be detected by consideration of all $k+1$ processes.

Let L_{pull} be the following language:

$$P ::= \mathbf{0} \mid n[\,P\,] \mid P \mid Q \mid \mathsf{pull}\, n.P \mid !\,P$$

Theorem 5.36 *Termination is decidable for* L_{pull}.

In proving the theorem we follow the same strategy as for Theorem 5.34. We first change from L_{pull} with full replication to L'_{pull} with replication just on pull and ambients. Next we define a non-standard reduction relation $\rightarrow^D$ which traps divergence finitely. We then show that $(L'_{\mathsf{pull}}, \rightarrow^D)$ is terminating, which gives us a decision procedure. As the development is very similar to that of Section 5.2, we omit most details and just mention a few points.

The rules for $\rightarrow^D$ are as in Section 5.2, except that we replace rules (In1)-(In4) by five variants of (Pull), and (InDiv1)-(InDiv6) by:

$$\begin{array}{lr}
\text{(PullDiv1)} & n[\,!\,\mathsf{pull}\, m.P \mid Q\,] \mid !\,m[\,R\,] \rightarrow^D \mathsf{DIV} \\
\text{(PullDiv2)} & !\,n[\,\mathsf{pull}\, m.P \mid Q\,] \mid !\,m[\,R\,] \rightarrow^D \mathsf{DIV} \\
\text{(PullDiv3)} & !\,n[\,!\,\mathsf{pull}\, m.P \mid Q\,] \mid !\,m[\,R\,] \rightarrow^D \mathsf{DIV} \\
\text{(PullDiv4)} & !\,n[\,\mathsf{pull}\, n.P \mid Q\,] \rightarrow^D \mathsf{DIV} \\
\text{(PullDiv5)} & !\,n[\,!\,\mathsf{pull}\, n.P \mid Q\,] \rightarrow^D \mathsf{DIV}
\end{array}$$

Matters proceed as in Section 5.2 until we reach the analogue of Lemma 5.29, which is proved rather differently:

Lemma 5.37 $(L'_{\mathsf{pull}}, \rightarrow^D_{top})$ *is terminating.*

PROOF. Take P_0 in L'_{pull}, and suppose that there is an infinite computation $P_0 \rightarrow^D_{top} \cdots \rightarrow^D_{top} P_i \rightarrow^D_{top} \cdots$. Then either (i) at least one top-level ambient performs infinitely many pulls, or (ii) at least one top-level replicated ambient performs infinitely many pulls.

If a single ambient performs infinitely many pulls, it can only do so because of a replicated capability $!\,\mathsf{pull}\, n$. Also, P_0 must have a top-level $!\,n[\,Q\,]$. Suppose that $!\,\mathsf{pull}\; n$ is first enabled in P_i. Then we have $P_i \rightarrow^D \mathsf{DIV}$ by (PullDiv1), and so $P_i \not\rightarrow^D_{top}$.

Suppose that a single replicated ambient, say $!\,m[\,P\,]$, performs infinitely many pulls. There must be some name n for which $!\,m[\,P\,]$ performs $\mathsf{pull}\, n$ infinitely often. But this is only possible if P_0 has a top-level $!\,n[\,Q\,]$ (which may of course be the same as $!\,m[\,P\,]$). This means that $P_0 \rightarrow^D \mathsf{DIV}$, using one of rules (PullDiv2)-(PullDiv5). Hence $P_0 \not\rightarrow^D_{top}$. □

As far as the analogue of Lemma 5.31 is concerned, the proof is much the same, though we note that there is a difference when it comes to analysing $\rightarrow^D_{top}$ reductions. With L'_{in}, a spun-off ambient could not perform an in, and hence could not have children. By contrast, with L'_{pull} spun-off ambients can pull in other ambients, and so can have children. If the spun-off ambient corresponds to $d' \in T$, where $(d, T) \in S_i$, we can add the degrees of its children to T, since we know that they are less than $d' < d$.

6 Conclusions and Future Work

The main contribution of this paper is to show that the open capability is not needed to obtain Turing completeness for pure Ambient Calculi. This implies that pure Boxed Ambients is Turing-complete.

We have sought to establish the minimality of the language L_{io} by showing that removing either in or out capabilities leads to a failure of Turing completeness in a rather dramatic fashion: every computation terminates.

A language very like L_{io} is studied in [23]. There it is shown that this language admits symmetric electoral systems, and also that any fragment of MA with this property must possess both in and out capabilities. It follows that there can be no encoding satisfying certain conditions of reasonableness from L_{io} into any fragment of MA not including both in and out capabilities.

We summarise our main contributions to understanding the computational strength of MA dialects in Figure 3 below. In the diagram we label each node with a language and with its strength. The languages all have full replication (where not stated otherwise) and are identified by the capabilities reported on the node. For example, $\mathsf{open}, \mathsf{in}, \mathsf{out}$ is pure public MA. A similar diagram holds for the results on PAC, with one exception: the language with push alone and replication just on push is not terminating (see Section 4.2), and so has decidable termination. In addition, a similar diagram holds for the results on SA, with two exceptions: (1) the language with out, $\overline{\mathsf{out}}$ alone and replication just on out, $\overline{\mathsf{out}}$ is not terminating (see Section 4.2), and so has decidable termination; and (2) it is an open question whether the language with in, $\overline{\mathsf{in}}$ and full replication has decidable termination (see Remark 5.35).

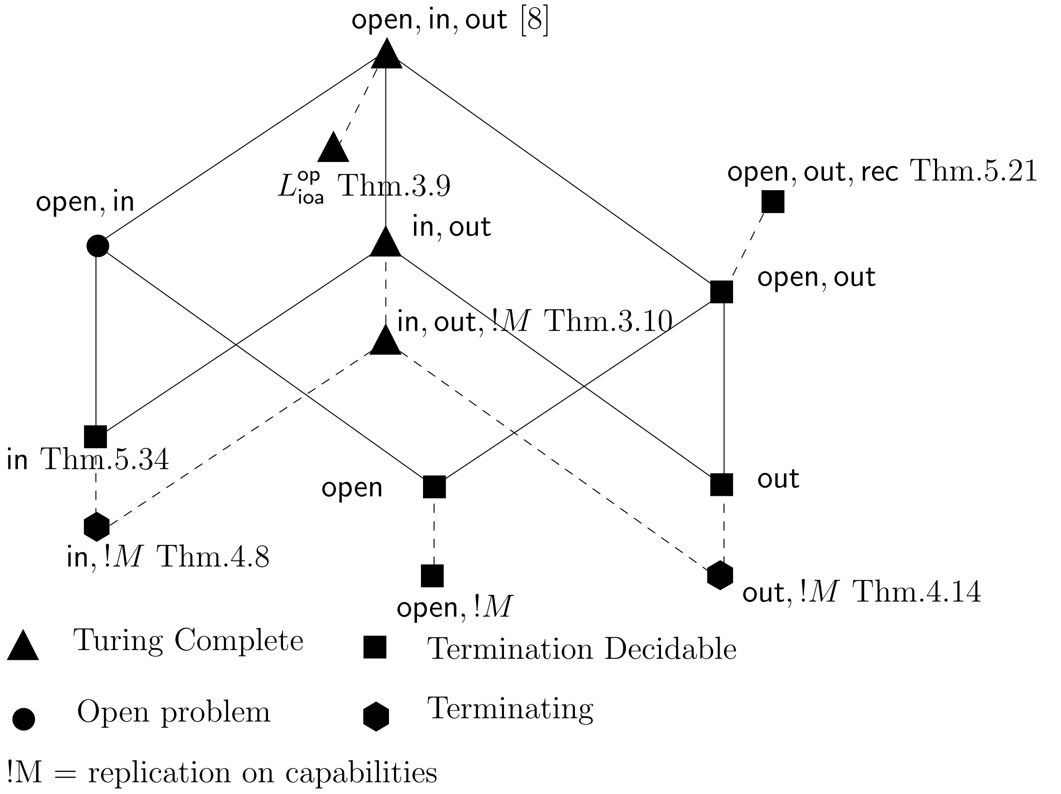

Fig. 3. Computational strength of some Ambient Calculi

We briefly mention some open questions/future work:

- As far as the study of the computational strength of fragments of pure Ambient Calculi is concerned, the major open question is the strength of the fragment with in and open capabilities (but not out). We conjecture that this fragment has decidable termination.
- We have seen that the language with in as its only capability and replication (but not restriction) has decidable termination (Theorem 5.34). It is an open question whether this is also the case when replication is replaced by restriction.
- The present work leads us to ask what might be a set of minimal constructs of AC capable of encoding regular expressions or context-free grammars.

Acknowledgements

We wish to thank Cristiano Calcagno, Philippa Gardner, Maria Grazia Vigliotti and Nobuko Yoshida for helpful discussions and encouragement. We also thank the anonymous referees for helpful comments. Maffeis was supported by a grant

from Microsoft Research, Cambridge.

A Encoding of CMs into $L^{\mathsf{op}}_{\mathsf{ioa}}$

We present an encoding of CMs into the language $L^{\mathsf{op}}_{\mathsf{ioa}}$ defined in Section 3.4:

$$P ::= \mathbf{0} \mid n[\,P\,] \mid P \mid Q \mid \mathsf{open}\, n.P \mid \mathsf{in}\, n.\mathbf{0} \mid \mathsf{out}\, n.\mathbf{0} \mid !\,\mathsf{open}\, n.P$$

Theorem 3.9 *$L^{\mathsf{op}}_{\mathsf{ioa}}$ is Turing-complete.*

PROOF. The proof of Turing completeness follows the structure of that of Theorem 3.8.

Numerals contain movement capabilities to interact with the instruction for decrement/jump, and each register contains a capability that will allow it to interact with both instructions:

$$\begin{aligned}\underline{0} &\overset{\mathrm{df}}{=} z[\,\mathsf{in}\, jz\,]\\ \underline{k+1} &\overset{\mathrm{df}}{=} s[\,\underline{k} \mid \mathsf{in}\, ds\,]\\ [\![R_j(k)]\!] &\overset{\mathrm{df}}{=} r_j[\,\mathsf{in}\, r_j \mid \underline{k}\,]\end{aligned}$$

The encoding is completely deterministic, since at each step only one reduction is possible. We define the encoding at the lth stage of an arbitrary configuration of CM:

$$[\![CM(i : k_0, \ldots, k_b)]\!]_l \overset{\mathrm{df}}{=} \mathsf{st}_i[\;] \mid \prod_{i \leq a} [\![I_i]\!]_l \mid \prod_{j \leq b} [\![R_j(k_j)]\!]$$

We now describe the encoding of the instructions. To increment a register r_j, we first make it enter in a dummy copy of itself which, once it acknowledges the presence of the register, moves into a skeleton containing the additional successor ambient to add. Once this dummy r_j is inside s, it is opened, the numeral is released inside the new s, and an acknowledgement ambient b is recognised both by the enclosing r_j, which creates its new capability $\mathsf{in}\, r_j$, and successively (ambient c) by the environment which releases the incremented register in the top level, along with the token for the continuation st_{i+1}.

$$\begin{aligned}
&[\![i : \mathsf{Inc}(j)]\!]_l \stackrel{\text{df}}{=} \\
&\qquad !\,\mathsf{open}\ \mathsf{st}_i.(r_j[\,\mathsf{open}\ r_j.(\mathsf{in}\ u \mid \mathsf{in}\ r_j \mid \mathsf{in}\ s)\,] \\
&\qquad \mid u[\,r_j[\,\mathsf{open}\ b.\mathsf{in}\ r_j \mid s[\,\mathsf{in}\ ds \mid \mathsf{open}\ r_j.b[\,\mathsf{out}\ s \mid c[\,\mathsf{out}\ r_j \mid \mathsf{out}\ u\,]\,]\,]\,]\,] \\
&\qquad \mid \mathsf{open}\ c.\mathsf{open}\ u.\mathsf{st}_{i+1}[\,])
\end{aligned}$$

Notice that the encoding of $i : \mathsf{Inc}(j)$ does not in fact depend on the step l of the computation, since there is no garbage (there will be garbage when we come to decrement/jump). The encoding satisfies:

$$\mathsf{st}_i[\,] \mid [\![i : \mathsf{Inc}(j)]\!]_l \mid [\![R_j(k)]\!] \Rightarrow \mathsf{st}_{i+1}[\,] \mid [\![i : \mathsf{Inc}(j)]\!]_{l+1} \mid [\![R_j(k+1)]\!]$$

The instruction for decrement/jump is complicated by the need to dispose of the jump branch if a decrement is executed, or the decrement branch if the register contains 0.

$$\begin{aligned}
[\![i : \mathsf{DecJump}(j, i')]\!]_l \stackrel{\text{df}}{=}\ & !\,\mathsf{open}\ \mathsf{st}_i.r_j[\,\mathsf{open}\ r_j.(DS(i) \mid JZ(i) \mid F(i) \mid \mathsf{in}\ r_j)\,] \\
& \mid CLR(i, ds) \mid CLR(i, jz) \\
& \mid GRB(i, ds, jump(i, l)) \mid GRB(i, jz, dec(i, l))
\end{aligned}$$

The strategy consists in opening the instruction trigger (st_i), inviting the register inside a dummy copy where it is opened and then having the numeral itself selecting either the $DS(i)$ or the $JZ(i)$ term according to its value. The selected term must make sure that the other one is disposed and processes $CLR(i, ds), CLR(i, jz)$ make sure (interacting with $F(i)$) that all the garbage is collected, and trigger the appropriate continuation.

Below, x and y are complementary syntactic macros, such that if $x = jz$ in a term, then $y = ds$ (and vice versa).

$$\begin{array}{rcl}
DS(i) & \overset{\text{df}}{=} & ds[\,\mathsf{open}\; s.DISP1(i, jz) \mid \mathsf{in}\; dds_i \mid \mathsf{in}\; b\,] \\
JZ(i) & \overset{\text{df}}{=} & jz[\,\mathsf{open}\; z.(DISP1(i, ds) \mid z[\,\mathsf{in}\; jz\,]) \mid \mathsf{in}\; djz_i \mid \mathsf{in}\; b\,] \\
F(i) & \overset{\text{df}}{=} & \mathsf{open}\; a.\mathsf{open}\; end.\mathsf{open}\; djz_i.\mathsf{open}\; dds_i.\mathsf{open}\; b \\
CLR(i, x) & \overset{\text{df}}{=} & !\,\mathsf{open}\; dx_i.a[\,\mathsf{in}\; r_j \mid DISP2(i, y)\,] \\
GRB(i, ds, n) & \overset{\text{df}}{=} & (b[\,\mathsf{open}\; s.DISP1(i, jz)\,])^n \\
GRB(i, jz, n) & \overset{\text{df}}{=} & (b[\,\mathsf{open}\; z.(DISP1(i, ds) \mid z[\,\mathsf{in}\; jz\,])\,])^n \\
DISP1(i, x) & \overset{\text{df}}{=} & dx_i[\,\mathsf{out}\; y \mid b[\,\mathsf{open}\; x.c[\,\mathsf{out}\; b\,]\,] \mid \mathsf{open}\; c.\mathsf{out}\; r_j\,] \\
DISP2(i, x) & \overset{\text{df}}{=} & dx_i[\,b[\,\mathsf{open}\; x.end[\,\mathsf{out}\; b \mid \mathsf{out}\; dx_i \mid dy_i[\,]\,] \mid ST(x)\,]\,] \\
ST(ds) & \overset{\text{df}}{=} & \mathsf{st}_{i+1}[\,\mathsf{out}\; r_j\,] \\
ST(jz) & \overset{\text{df}}{=} & \mathsf{st}_{i'}[\,\mathsf{out}\; r_j\,]
\end{array}$$

We follow step by step an example where decrement takes place. The case for jump is almost symmetric. The initial state is

$$\ldots \mid \mathsf{st}_i[\,] \mid [\![i : \mathsf{DecJump}(j, i')]\!]_l \mid [\![R_j(k+1)]\!] \mid \ldots .$$

After the first three steps we reach

$$\ldots \mid r_j[\,s[\,\underline{k} \mid \mathsf{in}\; ds\,] \mid DS(i) \mid JZ(i) \mid F(i) \mid \mathsf{in}\; r_j\,] \mid \ldots .$$

Now s enters ds, it is opened, and djz_i exits ds:

$$\begin{array}{l}
\ldots \mid r_j[\,ds[\,\underline{k} \mid \mathsf{in}\; dds_i \mid \mathsf{in}\; b\,] \\
\qquad\qquad \mid djz_i[\,b[\,\mathsf{open}\; jz.c[\,\mathsf{out}\; b\,]\,] \mid \mathsf{open}\; c.\mathsf{out}\; r_j\,] \mid \ldots\,] \mid \ldots
\end{array}$$

Ambient jz enters djz_i and b, gets opened, c leaves b, gets opened, and djz_i leaves r_j.

$$\ldots \mid r_j[\,ds[\ldots] \mid F(i) \mid \mathsf{in}\; r_j\,] \mid djz_i[\,b[\,\mathsf{open}\; z.(\ldots)\,]\,] \mid CLR(i, jz) \mid \ldots$$

Now djz_i is opened by $CLR(i, jz)$, a enters r_j and gets opened by $F(i)$ releasing $DISP2(i, ds)$ in r_j.

$$\ldots \mid r_j[\,ds[\ldots] \mid \mathsf{open}\; end.(\ldots) \mid \mathsf{in}\; r_j \mid DISP2(i, ds)\,] \mid GRB(i, jz, 1) \mid \ldots$$

Ambient ds now enters dds_i and b, gets opened, and ambient end exits to the top level in r_j.

$$\ldots \mid r_j[\,dds_i[\,b[\,\underline{k} \mid ST(ds)\,]\,]\,] \mid \mathsf{open}\; end.(\ldots) \mid \mathsf{in}\; r_j \mid end[\,djz_i[\;]\,]\,] \mid \ldots$$

Now end is opened, followed by djz_i, then dds_i, and finally b is opened, releasing the continuation, which exits r_j. Assuming that $dec(i,l) = m$, we have

$$\ldots \mid \mathsf{st}_{i+i}[\;] \mid r_j[\,\underline{k} \mid \mathsf{in}\; r_j\,] \mid GRB(i,jz,1) \mid GRB(i,jz,m) \mid \ldots$$

By definition, we have that $GRB(i,jz,1) \mid GRB(i,jz,m) = GRB(i,jz,m+1)$, and since a decrement has been executed $dec(i,l+1) = m+1$, and we conclude with

$$\ldots \mid \mathsf{st}_{i+i}[\;] \mid [\![i : \mathsf{DecJump}(j,k)]\!]_{l+1} \mid [\![R_j(k)]\!] \mid \ldots$$

□

B Encoding of CMs into L_{io}

We present an encoding of CMs into the language L_{io} defined in Section 3.5:

$$P ::= \mathbf{0} \;\mid\; n[\,P\,] \;\mid\; P \mid Q \;\mid\; \mathsf{in}\; n.P \;\mid\; \mathsf{out}\; n.P \;\mid\; !\,\mathsf{in}\; n.P \;\mid\; !\,\mathsf{out}\; n.P$$

Theorem 3.10 *L_{io} is Turing-complete.*

PROOF. (See first the sketch in Section 3.5.) We consider a particular CM called CM, with instructions $I_0, \ldots, I_a$ and registers $R_0, \ldots, R_b$. Let $CM(i : k_0, \ldots, k_b)$ represent CM when it is about to execute instruction i and storing k_j in register j ($j \leq b$). Let the (unique) finite or infinite computation of $CM = CM_0$ be $CM_0, CM_1, \ldots, CM_l, \ldots$, where $CM_l = CM(i_l : k_{0l}, \ldots, k_{bl})$.

Each register R_j ($j \leq b$) is encoded as an r_j ambient enclosing a numeral process $\underline{k}$ encoding the stored natural number k. Let the instructions I_i be numbered from 0 to a. The outer r_j ambient has the task of entering any st_i ambient ($i \leq a$). The first register R_0 is additionally allowed to enter st_{a+1}. This will allow R_0 to be conveyed back up to the top level to give the result of the computation.

In describing the encoding of the registers and instructions, we must take into account the fact that both the increment and the decrement/jump instructions will accumulate garbage each time they are used. We therefore parametrise our encoding by the index l of the stage we have reached in the computation. Let

- $\mathit{inc}(i, l)$ be the number of increments
- $\mathit{dec}(i, l)$ be the number of decrements
- $\mathit{dec}_s(i, l)$ be the number of decrements leaving the register contents non-zero
- $\mathit{dec}_z(i, l)$ be the number of decrements leaving the register contents zero
- $\mathit{jump}(i, l)$ be the number of jumps

performed by instruction i during the computation of CM up to, but not including, stage l. Clearly, $\mathit{dec}(i, l) = \mathit{dec}_s(i, l) + \mathit{dec}_z(i, l)$.

$$[\![R_0(k)]\!]_l \stackrel{\text{df}}{=} r_0[\,\underline{k}_l \mid \textstyle\prod_{i\le a+1} !\,\mathsf{in}\;\mathsf{st}_i\,]$$

$$[\![R_j(k)]\!]_l \stackrel{\text{df}}{=} r_j[\,\underline{k}_l \mid \textstyle\prod_{i\le a} !\,\mathsf{in}\;\mathsf{st}_i\,] \quad (1 \le j \le b)$$

Register 0 has special treatment to deal with finishing off the computation and making the contents available to any further computation. The numeral processes are defined as follows:

$$\underline{0}_l \stackrel{\text{df}}{=} z[\,IZ \mid D_t \mid (\mathit{increq}[\,!\,\mathsf{in}\;s.\mathsf{in}\;t\,])^{\mathit{inc}(i,l)}\,]$$

$$IZ \stackrel{\text{df}}{=} !\,\mathsf{in}\;s.\mathsf{in}\;t$$

$$D_t \stackrel{\text{df}}{=} !\,\mathsf{in}\;\mathit{dect}'.\mathsf{out}\;\mathit{dect}'.\mathsf{out}\;t.\mathsf{out}\;\mathit{dect}$$

Here IZ helps with increment, and D_t helps with decrement. The increq ambients build up as garbage inside $\underline{0}_l$ with each increment.

$$\underline{k+1}_l \stackrel{\text{df}}{=} s[\,DS \mid D_t \mid t[\,DT \mid D_s \mid \underline{k}_l\,]\,]$$

$$DS \stackrel{\text{df}}{=} \mathsf{in}\;\mathit{decs}$$

$$DT \stackrel{\text{df}}{=} \mathsf{in}\;\mathit{dect}$$

$$D_s \stackrel{\text{df}}{=} \mathsf{in}\;\mathit{decs}'.\mathsf{out}\;\mathit{decs}'.\mathsf{out}\;s.\mathsf{out}\;\mathit{decs}$$

The processes inside s and t help with decrement.

It is convenient to have a monitor process Mon which checks that all the registers and instructions have entered the st_i ambient to reach the current level.

$$Mon \stackrel{\text{df}}{=} m[\,\textstyle\prod_{i\le a} !\,\mathsf{in}\;\mathsf{st}_i.M_i\,]$$

$$M_i \stackrel{\text{df}}{=} \mathsf{in}\;p_0.\mathsf{out}\;p_0.\cdots\mathsf{in}\;p_a.\mathsf{out}\;p_a.\mathsf{in}\;r_0.\mathsf{out}\;r_0.\cdots.\mathsf{in}\;r_b.\mathsf{out}\;r_b.m_i[\,\mathsf{out}\;m\,]$$

Once the monitor has finished checking, it unleashes ambient m_i and instruction i is free to go ahead. Once st_i appears, the instructions and registers reach the next level in an indeterminate order. However, once the monitor has finished its check, the computation proceeds deterministically until execution of I_i is complete (except for a limited concurrency in the increment, noted below).

We now describe the encoding of the CM instructions. The process corresponding to instruction I_i $(i \leq a)$ is of the form

$$[\![I_i]\!]_l \stackrel{\text{df}}{=} p_i[\,(\prod_{i' \leq a} !\,\mathsf{in}\,\mathsf{st}_{i'}) \mid !\,\mathsf{in}\,m_i.\mathsf{out}\,m_i.P_i \mid G_{il}\,]$$

where P_i carries out the instruction, which is either increment or test and decrement or jump, and G_{il} is the garbage which accumulates during the computation up to stage l. The process P_i will first exit p_i and then enter the appropriate register r_j.

Once the computation is complete, the st_{a+1} ambient conveys R_0 back up to the top level using the following process:

$$F_{a+1} \stackrel{\text{df}}{=} \mathit{check}[\,\mathsf{in}\,r_0.\mathsf{out}\,r_0.\mathsf{out}\,\mathsf{st}_{a+1}\,] \mid \mathsf{in}\,\mathit{check}.\mathsf{out}\,\mathit{check}.(\prod_{i \leq a} !\,\mathsf{out}\,\mathsf{st}_i)$$

Thus $\mathsf{st}_{a+1}[\,F_{a+1}\,]$ first checks whether R_0 has entered, and then moves up to the top level. The *check* ambient is left behind as garbage. For $i \leq a$, the st_i ambient does nothing further once it has appeared at the current level; it is convenient to define $F_i \stackrel{\text{df}}{=} \mathbf{0}$ $(i \leq a)$.

Before giving the instruction and garbage processes P_i, G_{il} in detail, we complete the encoding of the CM. We capture the way that the computation moves down successive levels by the following contexts:

$$\mathcal{C}_0\{\bullet\} \stackrel{\text{df}}{=} \bullet$$
$$\mathcal{C}_{l+1}\{\bullet\} \stackrel{\text{df}}{=} \mathcal{C}_l\{\mathsf{st}_{i_l}[\,m_{i_l}[\,] \mid \bullet\,]\}$$

where i_l is the instruction performed at the lth stage. The overall encoding of the CM is:

$$[\![CM(i : k_0, \ldots, k_b)]\!]_l \stackrel{\text{df}}{=}$$
$$\mathcal{C}_l\{\mathsf{st}_i[\,!\,\mathsf{out}\,t.\mathsf{out}\,s \mid F_i\,] \mid \mathit{Mon} \mid (\textstyle\prod_{i \leq a}[\![I_i]\!]_l) \mid (\prod_{j \leq b}[\![R_j(k_j)]\!]_l)\}$$

The encoding of CM is $[\![CM]\!] \overset{\mathrm{df}}{=} [\![CM_0]\!]_0$. The encoded CM will go through successive stages $[\![CM_l]\!]_l$. We show that for each non-terminal stage l, $[\![CM_l]\!]_l \Rightarrow [\![CM_{l+1}]\!]_{l+1}$, and that $[\![CM_l]\!]_l$ is guaranteed to reach $[\![CM_{l+1}]\!]_{l+1}$. There are various cases according to whether we are dealing with increment, decrement or jump.

The increment instruction $i : \mathsf{Inc}(j)$ is carried out by an ambient $increq$ which leaves p_i and then penetrates to the core of the register r_j (inside z). Then st_{i+1} is unleashed, and leaves $increq$ and z. The new $s[\,t[\;]\,]$ then leaves st_{i+1}. Now z can enter s followed by t. We need to check that z has reached the core. So st_{i+1} enters s, t and finally z. Note that there is limited concurrency at this point between z entering s, t and st_{i+1} entering s, t. This does not cause a problem, as there is synchronisation when st_{i+1} enters z. Now the increment is complete, and st_{i+1} makes its way back out of r_j. At this point the next instruction is triggered.

$$\begin{aligned}
P_i &\overset{\mathrm{df}}{=} increq[\,\mathsf{out}\,p_i.\mathsf{in}\,r_j.(\,!\,\mathsf{in}\,s.\mathsf{in}\,t \mid \mathsf{in}\,z.IST)\,]\\
IST &\overset{\mathrm{df}}{=} \mathsf{st}_{i+1}[\,\mathsf{out}\,increq.\mathsf{out}\,z.(s[\,\mathsf{out}\,\mathsf{st}_{i+1}.(DS \mid D_t \mid t[\,DT \mid D_s\,])\,] \mid IA)\,]\\
IA &\overset{\mathrm{df}}{=} \mathsf{in}\,s.\mathsf{in}\,t.\mathsf{in}\,z.\mathsf{out}\,z.(\,!\,\mathsf{out}\,t.\mathsf{out}\,s \mid \mathsf{out}\,r_j.F_{i+1})
\end{aligned}$$

Note that $increq[\,!\,\mathsf{in}\,s.\mathsf{in}\,t\,]$ is left as garbage at the core of the register inside z. There is no garbage inside p_i, and so we define $G_{il} \overset{\mathrm{df}}{=} \mathbf{0}$.

In order to implement the instruction $i : \mathsf{DecJump}(j, i')$, we must test for whether the register R_j is zero or nonzero. This is done by the following process:

$$\begin{aligned}
P_i &\overset{\mathrm{df}}{=} test[\,\mathsf{out}\,p_i.\mathsf{in}\,r_j.(Q_z \mid Q_s)\,]\\
Q_z &\overset{\mathrm{df}}{=} \mathsf{in}\,z.\mathsf{out}\,z.\mathsf{out}\,r_j.\mathsf{in}\,p_i.\mathsf{st}_{i'}[\,\mathsf{out}\,test.\mathsf{out}\,p_i.F_{i'}\,]\\
Q_s &\overset{\mathrm{df}}{=} \mathsf{in}\,s.\mathsf{out}\,s.\mathsf{out}\,r_j.\mathsf{in}\,p_i.P'_i
\end{aligned}$$

The $test$ ambient enters r_j. If it detects z it leaves the register, re-enters p_i and unleashes instruction i'. The process $test[\,Q_s\,]$ remains as garbage inside p_i. Otherwise $test$ detects s, leaves the register, re-enters p_i and unleashes process P'_i, which performs the decrement of the register before proceeding to instruction $i+1$. The process $test[\,Q_z\,]$ remains as garbage inside p_i.

Decrement is performed in two stages: first strip off the outermost s, and then strip off t.

$$P'_i \overset{\mathrm{df}}{=} decs[\,\mathsf{out}\,test.\mathsf{out}\,p_i.\mathsf{in}\,r_j.(decs'[\,\mathsf{in}\,s\,] \mid \mathsf{in}\,t.\mathsf{out}\,t.\mathsf{out}\,r_j.\mathsf{in}\,p_i.P''_i)\,]$$

To start with, $decs$ goes to the top level inside r_j. Suppose the register contains $\underline{k+1}_l$. The portion of interest of the CM process is:

$$\ldots r_j[\, decs[\, decs'[\,\mathsf{in}\; s\,] \mid \mathsf{in}\; t.(\ldots)\,] \mid s[\, DS \mid D_t \mid t[\, DT \mid D_s \mid \underline{k}_l\,]\,]\,] \ldots$$

Then the whole contents of the register enter using DS. Then $decs'$ enters s, which activates D_s, leading to t going to the top level inside r_j.

$$\ldots r_j[\, decs[\,\mathsf{in}\; t.(\ldots) \mid s[\, decs'[\,] \mid D_t\,]\,] \mid t[\, DT \mid \underline{k}_l\,]\,] \ldots$$

This is detected by $decs$, which exits r_j, enters p_i and unleashes P_i''. The first stage is completed. The process $decs[\, s[\, D_t \mid decs'[\,]\,]\,]$ remains as garbage inside p_i.

Now we must strip off the outermost t to complete the decrement. The procedure is roughly the same, with s and t swapped.

$$\begin{aligned}
P_i'' &\stackrel{\mathrm{df}}{=} dect[\,\mathsf{out}\; decs.\mathsf{out}\; p_i.\mathsf{in}\; r_j.(dect'[\,\mathsf{in}\; t\,] \mid Q_s' \mid Q_z')\,] \\
Q_s' &\stackrel{\mathrm{df}}{=} \mathsf{in}\; z.\mathsf{out}\; z.P_i''' \\
Q_z' &\stackrel{\mathrm{df}}{=} \mathsf{in}\; s.\mathsf{out}\; s.P_i''' \\
P_i''' &\stackrel{\mathrm{df}}{=} \mathsf{out}\; r_j.\mathsf{in}\; p_i.\mathsf{st}_{i+1}[\,\mathsf{out}\; dect.\mathsf{out}\; p_i.(\,!\,\mathsf{out}\; t.\mathsf{out}\; s \mid F_{i+1})\,]
\end{aligned}$$

The ambient $dect$ enters the register:

$$\ldots r_j[\, dect[\, dect'[\,\mathsf{in}\; t\,] \mid Q_s' \mid Q_z' \mid t[\, DT \mid \underline{k}_l\,]\,]\,] \ldots$$

Now t enters $dect$, and $dect'$ enters t:

$$\ldots r_j[\, dect[\, Q_s' \mid Q_z' \mid t[\, dect'[\,] \mid \underline{k}_l\,]\,]\,] \ldots$$

The numeral $\underline{k}_l$ uses D_t to exit t and $dect$:

$$\ldots r_j[\, dect[\, Q_s' \mid Q_z' \mid t[\, dect'[\,]\,]\,] \mid \underline{k}_l\,] \ldots$$

The end of the decrement is signalled by st_{i+1} appearing at the level of p_i and r_j. Depending on whether the decremented register is zero or non-zero, we have either $dect[\, Q_s' \mid t[\, dect'[\,]\,]\,]$ or $dect[\, Q_z' \mid t[\, dect'[\,]\,]\,]$ as extra garbage inside p_i. We therefore define G_{il} to be

$$\begin{aligned}
&(test[\, Q_s\,])^{jump(i,l)} \mid (test[\, Q_z\,] \mid decs[\, s[\, D_t \mid decs'[\,]\,]\,])^{dec(i,l)} \mid \\
&(dect[\, Q_z' \mid t[\, dect'[\,]\,]\,])^{dec_s(i,l)} \mid (dect[\, Q_s' \mid t[\, dect'[\,]\,]\,])^{dec_z(i,l)}
\end{aligned}$$

It can be verified that all garbage can take no further part in the computation.

At the end of the computation (if it terminates) a st_{a+1} ambient is unleashed (recall that the last valid instruction number is a). This ambient then appears at the top level containing R_0. Thus the CM terminates iff $[\![CM]\!] \Downarrow \mathsf{st}_{a+1}$. This establishes that the weak barb relation is undecidable, and that having a non-terminating computation is undecidable.

To fulfil Criterion 3.1 we must ensure that R_0 is able to be used as input by further computations. The problem is that the encoding of the register makes explicit use of the list of instructions in order to allow it to enter st_i $(i \leq a+1)$. We resolve this problem by starting any subsequent computation by first transferring R_0 into a new first register which is suited to the new instruction list. This can be done by three CM instructions, as follows.

Let the new CM be CM'. With appropriate renumbering, its program proper uses registers numbered $1, 2, \ldots b'$ (with the result being placed in register 1) and its instructions are numbered $a+1, \ldots, a'$, with $a+1, a+2, a+3$ copying the contents of register 0 into register 1, and $a+4$ being the index of the first true instruction of CM'. We also assume a register $R_{b'+1}$ with contents set to 0 (this is used in instruction I_{a+3}).

$$a+1 : \mathsf{DecJump}(0, a+4)$$

$$a+2 : \mathsf{Inc}(1)$$

$$a+3 : \mathsf{DecJump}(b'+1, a+1)$$

$$a+4 : \text{Start of } CM' \text{ proper}$$

We adjust the definition of R_0 in CM so that it can take part in instructions I_{a+1}, I_{a+2} and I_{a+3}:

$$[\![R_0(k)]\!]_l \overset{\text{df}}{=} r_0[\, \underline{k}_{\,l} \mid \prod_{i \leq a+3} !\,\mathsf{in}\ \mathsf{st}_i \,]$$

We define the monitor process Mon of CM' in such a way that the old R_0 is not expected to travel beyond instruction $a+3$; we omit the details.

Strictly speaking, we should have taken all this into account in our definitions of the encoding, but it seemed clearer not to do this.

One can adapt the above encoding to ensure that there are no continuations after the "out" capabilities. An essential difference is that it is not clear how to adapt the monitor process, which is therefore dispensed with. Thus there will be concurrency, in that the registers and instructions will make their way downwards at different rates, but this does not lead to any erroneous computations. Similar considerations apply to the increment: the process has to be changed to a more nondeterministic one, though again without any erroneous computations. □

C Encoding of CMs into L_{pp}

We present an encoding of CMs into the language L_{pp} defined in Section 3.5:

$$P ::= \mathbf{0} \mid n[\,P\,] \mid P \mid Q \mid \mathsf{push}\, n.P \mid \mathsf{pull}\, n.P \mid !\,\mathsf{push}\, n.P \mid !\,\mathsf{pull}\, n.P$$

Theorem 3.13 *L_{pp} is Turing-complete.*

PROOF. We consider a particular CM called CM, with instructions $I_0, \ldots, I_a$ and registers $R_0, \ldots, R_b$. Let $CM(i : k_0, \ldots, k_b)$ represent CM when it is about to execute instruction i and storing k_j in register j ($j \leq b$). Let the (unique) finite or infinite computation of $CM = CM_0$ be $CM_0, \ldots, CM_l, \ldots$, where $CM_l = CM(i_l : k_{0l}, \ldots, k_{bl})$.

We shall describe how registers are encoded, followed by the same for instructions. Then we shall describe how the encoded CM operates in detail. In describing the encoding of the registers and instructions, we must take into account the fact that both the increment and the decrement/jump instructions will accumulate garbage each time they are used. We therefore parametrise our encoding by the index l of the stage we have reached in the computation. Let

- $inc(i, l)$ be the number of increments
- $dec_s(i, l)$ be the number of decrements leaving the register contents non-zero
- $dec_z(i, l)$ be the number of decrements leaving the register contents zero
- $jump(i, l)$ be the number of jumps

performed by instruction i during the computation of CM up to, but not including, stage l.

Zero and successor registers with their contents are encoded as follows:

$$\begin{aligned}
[\![R_j(0)]\!]_l &\overset{\mathrm{df}}{=} z_j[\,(increq_j[\,])^{inc(i,l)} \mid !\,\mathsf{pull}\, increq_j.\\
&\qquad (\mathsf{push}\, s_j \mid s_j[\,SZ_j \mid SD_j \mid I_j \mid t_j[\,TZ_j \mid TD_j \mid I_j\,]\,])\,]\\
[\![R_j(k+1)]\!]_l &\overset{\mathrm{df}}{=} s_j[\,SD_j \mid I_j \mid t_j[\,TD_j \mid I_j \mid [\![R_j(k)]\!]_l\,]\,]
\end{aligned}$$

Thus incrementing a register by 1 involves adding two new surrounding ambients s_j, t_j. These will actually be added to the core of the register process, immediately round the central z_j ambient, when a request is received (an $increq_j$ ambient is detected). The auxiliary t_j ambients are introduced to help in handling decrements.

$$\begin{aligned} SZ_j &\overset{\text{df}}{=} \mathsf{pull}\ z_j.\mathsf{push}\ incack_j \\ TZ_j &\overset{\text{df}}{=} \mathsf{pull}\ z_j.(\mathsf{push}\ incack_j \mid incack_j[\,]) \end{aligned}$$

The I_j process pulls $increq_j[\,]$ inwards towards the core, and pushes the acknowledgement $incack_j[\,]$ out towards the top level:

$$I_j \overset{\text{df}}{=} !\,\mathsf{pull}\ increq_j.\mathsf{push}\ incack_j$$

The SD_j and TD_j processes help in decrementing a nonzero register:

$$\begin{aligned} SD_j &\overset{\text{df}}{=} \mathsf{pull}\ u_j.\mathsf{push}\ t_j \\ TD_j &\overset{\text{df}}{=} \mathsf{pull}\ decreq_j.(TDS_j \mid TDZ_j) \\ TDS_j &\overset{\text{df}}{=} \mathsf{push}\ s_j.(\mathsf{push}\ decack_j \mid decack_j[\,]) \\ TDZ_j &\overset{\text{df}}{=} \mathsf{push}\ z_j.(\mathsf{push}\ decack_j \mid decack_j[\,]) \end{aligned}$$

We now turn to the instructions. The ith instruction is activated when a $\mathsf{st}_i[\,]$ ambient appears at the top level.

(1) Increment. The encoded instruction $[\![i : \mathsf{Inc}(j)]\!]_l$ is:

$$\begin{aligned} &p_i[\ !\,\mathsf{pull}\ \mathsf{st}_i.(increq_j[\,] \mid \mathsf{push}\ increq_j.\mathsf{pull}\ incack_j.(\mathsf{push}\ \mathsf{st}_{i+1} \mid \mathsf{st}_{i+1}[\,])) \\ &\mid (GI_{ij})^{inc(i,l)}\,] \end{aligned}$$

where $GI_{ij} \overset{\text{df}}{=} \mathsf{st}_i[\,] \mid incack_j[\,]$ is the garbage which accumulates with each increment.

(2) Test and decrement or jump. $[\![i : \mathsf{DecJump}(j, i')]\!]_l$ is:

$$\begin{aligned} &p_i[\ !\,\mathsf{pull}\ \mathsf{st}_i.(\mathsf{push}\ test \mid test[Testz_j \mid Tests_j]) \mid !\,FZ_{ji'} \mid !\,FS_{ij} \\ &\mid (GJ_{ij})^{jump(i,l)} \mid (GDS_{ij})^{dec_s(i,l)} \mid (GDZ_{ij})^{dec_z(i,l)}\,] \end{aligned}$$

where

$$\begin{aligned}
Testz_j &\stackrel{\mathrm{df}}{=} \mathsf{pull}\, z_j.\mathsf{push}\, z_j.(\mathsf{push}\, tested \mid tested[\,Testedz_j\,]) \\
Tests_j &\stackrel{\mathrm{df}}{=} \mathsf{pull}\, s_j.\mathsf{push}\, s_j.(\mathsf{push}\, tested \mid tested[\,Testeds_j\,]) \\
Testedz_j &\stackrel{\mathrm{df}}{=} \mathsf{pull}\, test.(\mathsf{push}\, donez_j \mid donez_j[\,]) \\
Testeds_j &\stackrel{\mathrm{df}}{=} \mathsf{pull}\, test.(\mathsf{push}\, dones_j \mid dones_j[\,]) \\
FZ_{ji'} &\stackrel{\mathrm{df}}{=} \mathsf{pull}\, donez_j.\mathsf{pull}\, tested.(\mathsf{push}\, \mathsf{st}_{i'} \mid \mathsf{st}_{i'}[\,]) \\
FS_{ij} &\stackrel{\mathrm{df}}{=} \mathsf{pull}\, dones_j.\mathsf{pull}\, tested.(FD_{ij} \mid decreq_j[\,DR_j\,]) \\
FD_{ij} &\stackrel{\mathrm{df}}{=} \mathsf{push}\, decreq_j.\mathsf{pull}\, decack_j.\mathsf{pull}\, t_j.(\mathsf{push}\, \mathsf{st}_{i+1} \mid \mathsf{st}_{i+1}[\,]) \\
DR_j &\stackrel{\mathrm{df}}{=} u_j[\,] \mid \mathsf{pull}\, s_j.\mathsf{push}\, t_j
\end{aligned}$$

Garbage can accumulate in three different ways, depending on whether the register contents are zero (giving a jump), or nonzero (giving a decrement where the new contents may be either zero or a successor):

$$\begin{aligned}
GJ_{ij} &\stackrel{\mathrm{df}}{=} \mathsf{st}_i[\,] \mid donez_j[\,] \mid tested[\,test[\,Tests_j\,]\,] \\
GDZ_{ij} &\stackrel{\mathrm{df}}{=} \mathsf{st}_i[\,] \mid dones_j[\,] \mid tested[\,test[\,Testz_j\,]\,] \\
&\qquad \mid decack_j[\,] \mid t_j[\,decreq_j[\,s_j[\,u_j[\,] \mid I_j\,]\,]\mid TDS_j \mid I_j\,] \\
GDS_{ij} &\stackrel{\mathrm{df}}{=} \mathsf{st}_i[\,] \mid dones_j[\,] \mid tested[\,test[\,Testz_j\,]\,] \\
&\qquad \mid decack_j[\,] \mid t_j[\,decreq_j[\,s_j[\,u_j[\,] \mid I_j\,]\,]\mid TDZ_j \mid I_j\,]
\end{aligned}$$

We define:

$$\llbracket CM(i : k_0, \ldots, k_b)\rrbracket_l \stackrel{\mathrm{df}}{=} \mathsf{st}_i[\,] \mid (\prod_{i \leq a} \llbracket I_i \rrbracket_l) \mid (\prod_{j \leq b} \llbracket R_j(k_j)\rrbracket_l)$$

The encoding of CM is $\llbracket CM \rrbracket \stackrel{\mathrm{df}}{=} \llbracket CM_0 \rrbracket_0$. The encoded CM will go through successive stages $\llbracket CM_l \rrbracket_l$. We show that for each non-terminal stage l, $\llbracket CM_l \rrbracket_l \Rightarrow \llbracket CM_{l+1} \rrbracket_{l+1}$, and that $\llbracket CM_l \rrbracket_l$ is guaranteed to reach $\llbracket CM_{l+1} \rrbracket_{l+1}$. Computation is entirely deterministic. There are various cases, depending on the kind of instruction.

First consider the execution of $\llbracket i : \mathsf{Inc}(j) \rrbracket_l$. Starting from

$$\mathsf{st}_i[\,] \mid \llbracket i : \mathsf{Inc}(j) \rrbracket_l \mid \llbracket R_j(k) \rrbracket_l$$

the instruction is activated (ambient p_i), and the $increq_j[\,]$ ambient is pushed to the top level:

$$\llbracket i : \mathsf{Inc}(j) \rrbracket_l \mid p_i[\,\mathsf{st}_i[\,] \mid \mathsf{pull}\, incack_j.(\ldots) \mid \ldots\,] \mid increq_j[\,] \mid \llbracket R_j(k) \rrbracket_l$$

Then the $increq_j[\,]$ ambient is pulled into the core of the register process, where it is added to the accumulated garbage. This leads to an s_j ambient being pushed out of z_j.

$$\ldots z_j[\,(increq_j[\,])^{inc(i,l+1)} \mid !\,\mathsf{pull}\; increq_j.(\ldots)\,] \mid s_j[\,SZ_j \mid \ldots]\ldots$$

Then z_j is pulled into s_j followed by t_j, so that the register is incremented.

$$\begin{aligned}\ldots s_j[\,&\mathsf{push}\; incack_j \mid SD_j \mid I_j \\ &\mid t_j[\,\mathsf{push}\; incack_j \mid incack_j[\,] \mid I_j \mid z_j[\ldots]\,]\,]\ldots\end{aligned}$$

The acknowledgement $incack_j[\,]$ is then pushed out to the top level, where it is pulled in by p_i, which then activates the next instruction by pushing out $\mathsf{st}_{i+1}[\,]$. The garbage $\mathsf{st}_i[\,] \mid incack_j[\,]$ (i.e. GI_{ij}) is left inside p_i, where it is added to the accumulated garbage. We now have

$$\mathsf{st}_{i+1}[\,] \mid [\![i : \mathsf{Inc}(j)]\!]_{l+1} \mid [\![R_j(k+1)]\!]_{l+1}$$

We now consider the execution of $[\![i : \mathsf{DecJump}(j,i')]\!]_l$. Starting from

$$\mathsf{st}_i[\,] \mid [\![i : \mathsf{DecJump}(j,i')]\!]_l \mid [\![R_j(k)]\!]_l$$

the instruction is activated (ambient p_i), and the *test* ambient is sent out to test whether k is zero or non-zero.

$$\ldots p_i[\mathsf{st}_i[\,] \mid !\,FZ_{ji'} \mid !\,FS_{ij} \mid \ldots] \mid test[\,Testz_j \mid Tests_j\,] \mid [\![R_j(k)]\!]_l$$

Once it has done the test it produces ambient *tested*, which signals the result to P_i by producing either $donez_j$ or $dones_j$, depending on whether k is zero or nonzero. There are now two possibilities, depending on whether k is zero or nonzero.

1. k is zero. Then $FZ_{ji'}$ enables p_i to pull in $testz_j$ and *tested*.

$$p_i[\,\mathsf{st}_i[\,] \mid donez_j[\,] \mid tested[\,test[\,Tests_j\,]\,] \mid \mathsf{push}\;\mathsf{st}_{i'} \mid \mathsf{st}_{i'}[\,] \mid \ldots]$$

Then p_i pushes out ambient $\mathsf{st}_{i'}$ to trigger the next instruction. (In the case that $i' = i$ there is a choice of ambients to push out, but this does not affect the determinism of the computation in any significant way.) The process

$$\mathsf{st}_i[\,] \mid donez_j[\,] \mid tested[\,test[\,Tests_j\,]\,]$$

(i.e. GJ_{ij}) is added to the accumulated garbage. We are left with

$$\mathsf{st}_{i'}[\,] \mid [\![i : \mathsf{DecJump}(j, i')]\!]_{l+1} \mid [\![R_j(k)]\!]_{l+1}\,.$$

2. k is nonzero. Then FS_{ij} enables p_i to pull in $tests_j$ and $tested$.

$$p_i[\,\mathsf{st}_i[\,] \mid dones_j[\,] \mid tested[\,test[\,Testz_j\,]\,] \mid FD_{ij} \mid decreq_j[\,DR_j\,] \mid \ldots]$$

Then p_i pushes out ambient $decreq_j$ to carry out the decrement. Then $decreq_j$ pulls in s_j (the entire register).

$$\begin{aligned} &p_i[\,\mathsf{st}_i[\,] \mid dones_j[\,] \mid tested[\,test[\,Testz_j\,]\,] \mid \mathsf{pull}\; decack_j.(\ldots) \mid \ldots] \\ &\qquad \mid decreq_j[\,u_j[\,] \mid \mathsf{push}\; t_j \mid s_j[\,SD_j \mid I_j \mid t_j[\,TD_j \mid I_j \mid [\![R_j(k-1)]\!]_l\,]\,]\,] \end{aligned}$$

Now s_j can pull in u_j and push out t_j. Then $decreq_j$ pushes t_j out to the top level, which enables t_j to detect it is at the top level by pulling in $decreq_j$.

$$\begin{aligned} &p_i[\,\mathsf{st}_i[\,] \mid dones_j[\,] \mid tested[\,test[\,Testz_j\,]\,] \mid \mathsf{pull}\; decack_j.(\ldots) \mid \ldots] \\ &\qquad \mid t_j[\,decreq_j[\,s_j[\,u_j[\,] \mid I_j\,]\,] \mid TDS_j \mid TDZ_j \mid I_j \mid [\![R_j(k-1)]\!]_l\,] \end{aligned}$$

Now t_j pushes out the decremented register—with outermost ambient either s_j or z_j, depending on the the value of k—and then signals completion of the decrement by pushing out $decack_j[\,]$. We illustrate the case when $k-1>0$:

$$\begin{aligned} &p_i[\,\mathsf{st}_i[\,] \mid dones_j[\,] \mid tested[\,test[\,Testz_j\,]\,] \mid \mathsf{pull}\; decack_j.(\ldots) \mid \ldots] \\ &\qquad \mid decack_j[\,] \mid t_j[\,decreq_j[\,s_j[\,u_j[\,] \mid I_j\,]\,] \mid TDZ_j \mid I_j\,] \mid [\![R_j(k-1)]\!]_l \end{aligned}$$

Then $decack_j$ is detected by p_i, which pulls in the left-over t_j, and activates the next instruction $i+1$. The garbage accumulates as either GDS_{ij} or GDZ_{ij}. We are left with

$$\mathsf{st}_{i+1}[\,] \mid [\![i : \mathsf{DecJump}(j, i')]\!]_{l+1} \mid [\![R_j(k-1)]\!]_{l+1}\,.$$

Finally, we see that if CM_L is terminal (so $i_L = a+1$) then $[\![CM_L]\!]_L$ has no reductions. $[\![CM_L]\!]_L$ displays barb st_{a+1} to indicate termination. The result of the computation, stored in register 0, is usable by subsequent computations. On the other hand, if CM does not terminate, then neither does $[\![CM]\!]$, and the barb st_{a+1} will never appear. There are no "bad" computations, i.e. ones which halt in a non-final state, diverge, or produce unintended behaviour. We have a encoding which shows Turing completeness, and also undecidability of termination and of weak barbs. □

References

[1] J. Bergstra, J.-W. Klop, Process algebra for synchronous communication, Information and Computation 60 (1984) 109–137.

[2] I. Boneva, J.-M. Talbot, When ambients cannot be opened, in: Proceedings of FoSSaCS 2003, Vol. 2620 of Lecture Notes in Computer Science, Springer-Verlag, 2003, pp. 169–184, full version to appear in Theoretical Computer Science.

[3] M. Bugliesi, G. Castagna, S. Crafa, Access control for mobile agents: the calculus of Boxed Ambients, ACM Transactions on Programming Languages and Systems 26 (1) (2004) 57–124.

[4] M. Bugliesi, S. Crafa, M. Merro, V. Sassone, Communication and mobility control in boxed ambients, Information and Computation, to appear.

[5] M. Bugliesi, S. Crafa, M. Merro, V. Sassone, Communication interference in mobile boxed ambients, in: Proceedings of FSTTCS '02, Vol. 2556 of Lecture Notes in Computer Science, Springer-Verlag, 2002, pp. 71–84, full version appears as [4].

[6] N. Busi, M. Gabbrielli, G. Zavattaro, Replication vs. recursive definitions in channel based calculi, in: Proceedings of ICALP '03, Vol. 2719 of Lecture Notes in Computer Science, Springer-Verlag, 2003, pp. 133–144.

[7] N. Busi, M. Gabbrielli, G. Zavattaro, Comparing recursion, replication, and iteration in process calculi, in: Proceedings of ICALP '04, Lecture Notes in Computer Science, Springer-Verlag, 2004, to appear.

[8] N. Busi, G. Zavattaro, On the expressive power of movement and restriction in pure mobile ambients, Theoretical Computer Science, in press June 2004.

[9] L. Cardelli, A. Gordon, Mobile ambients, Theoretical Computer Science 240 (1) (2000) 177–213.

[10] S. Crafa, M. Bugliesi, G. Castagna, Information flow security in boxed ambients, in: Proceedings of F-WAN: Workshop on Foundations of Wide-area Network Computing, Malaga, July 2002, Vol. 66 of Electronic Notes in Theoretical Computer Science, Elsevier, 2002.

[11] N. Dershowitz, Z. Manna, Proving termination with multiset orderings, Communications of the Association for Computing Machinery 22 (8) (1979) 465–476.

[12] A. Finkel, P. Schnoebelen, Well-structured transition systems everywhere!, Theoretical Computer Science 256 (2001) 63–92.

[13] L. Guan, Y. Yang, J. You, Making ambients more robust, in: Proceedings of the International Conference on Software: Theory and Practice, Beijing, China, August 2000, 2000, pp. 377–384.

[14] D. Hirschkoff, E. Lozes, D. Sangiorgi, Separability, expresssiveness, and decidability in the ambient logic, in: Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science, LICS 2002, IEEE Computer Society Press, 2002, pp. 423–432.

[15] F. Levi, D. Sangiorgi, Mobile safe ambients, ACM Transactions on Programming Languages and Systems 25 (1) (2003) 1–69.

[16] S. Maffeis, I. Phillips, On the computational strength of pure ambient calculi, in: Proceedings of Express: Workshop on expressiveness in concurrency, Marseille, September 2003, Electronic Notes in Theoretical Computer Science, Elsevier, to appear.

[17] M. Merro, M. Hennessy, Bisimulation congruences in safe ambients (extended abstract), in: Proceedings of the 29th Annual Symposium on Principles of Programming Languages, POPL '02, ACM, 2002, pp. 71–80.

[18] M. Merro, V. Sassone, Typing and subtyping mobility in boxed ambients, in: Proceedings of CONCUR '02, Vol. 2421 of Lecture Notes in Computer Science, Springer-Verlag, 2002, pp. 304–320.

[19] R. Milner, Communication and Concurrency, Prentice-Hall, 1989.

[20] M. Minsky, Computation: finite and infinite machines, Prentice-Hall, 1967.

[21] A. Phillips, N. Yoshida, S. Eisenbach, A distributed abstract machine for boxed ambient calculi, in: Proceedings of ESOP 2004, Vol. 2986 of Lecture Notes in Computer Science, Springer-Verlag, 2004, pp. 155–170.

[22] I. Phillips, M. Vigliotti, On reduction semantics for the push and pull ambient calculus, in: Proceedings of IFIP International Conference on Theoretical Computer Science (TCS 2002), IFIP 17th World Computer Congress, August 2002, Montreal, Kluwer, 2002, pp. 550–562.

[23] I. Phillips, M. Vigliotti, Electoral systems in ambient calculi, in: Proceedings of FoSSaCS 2004, Vol. 2987 of Lecture Notes in Computer Science, Springer-Verlag, 2004, pp. 408–422.

[24] D. Sangiorgi, Extensionality and intensionality of the ambient logics, in: Proceedings of POPL '01, ACM, 2001, pp. 4–17.

[25] D. Sangiorgi, D. Walker, The π-calculus, Cambridge University Press, 2001.

[26] J. Shepherdson, J. Sturgis, Computability of recursive functions, Journal of the Association for Computing Machinery 10 (1963) 217–255.

[27] D. Teller, P. Zimmer, D. Hirschkoff, Using ambients to control resources, in: Proceedings of CONCUR 2002, Vol. 2421 of Lecture Notes in Computer Science, Springer-Verlag, 2002, pp. 288–303, full version to appear in International Journal of Information Security.

[28] P. Zimmer, On the expressiveness of pure safe ambients, Mathematical Structures in Computer Science 13 (5) (2003) 721–770.