On the Computational Strength of Pure Ambient Calculi [1]

Sergio Maffeis and Iain Phillips [2,3,4]

Department of Computing
Imperial College London

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.

1 Introduction

Since its introduction in 1998, Cardelli and Gordon's calculus of Mobile Ambients (MA) [5,6] 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 Ambients (SA) [13,14], Robust Ambients (ROAM) [10], Safe Ambients with Passwords (SAP) [16], the Push and Pull Ambient Calculus (PAC) [20], Controlled Ambients (CA) [23], and the version of Boxed Ambients [2] with passwords (NBA) [3]. We shall use the term Ambient Calculus (AC) to refer to all of these variants.

[1] A shortened version of this report appears in EXPRESS 2003 [15].

[2] We wish to thank Cristiano Calcagno, Philippa Gardner, Maria Grazia Vigliotti and Nobuko Yoshida for helpful discussions and encouragement.

[3] Email: {maffeis,iccp}@doc.imperial.ac.uk

[4] Maffeis was supported by a grant from Microsoft Research, Cambridge.

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 pi-calculus into Mobile Ambients [6]. The encoding makes use of the communication primitives in the Ambient Calculus. However Cardelli and Gordon also encoded Turing machines directly into the *pure* AC, where there is no communication. (Incidentally, Zimmer [24] subsequently encoded the synchronous pi-calculus into pure Mobile Safe Ambients [13,14].)

Busi and Zavattaro [4] showed how to encode counter machines into pure Mobile Ambients without restriction. Independently, Hirschkoff, Lozes and Sangiorgi [11] encoded Turing machines into the same subcalculus. 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 [2,17,3,7]. They advocate communication between ambients where one is contained in the other, rather than the same-ambient communication of Mobile Ambients. A similar model of communication is employed in [19].

We give an encoding of counter machines into pure MA without restriction, and without the open capability (Theorem 3.6), 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 pi-calculus into Boxed Ambients with communication is given in [2]).

Two different kinds of ambient movement were identified by Cardelli and Gordon [6]: 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 [20]. 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) [13,14], 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) [10] 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.

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 MA with the open capability, but without movement capabilities or restriction, it is decidable whether a given process has a non-terminating computation [4]. We show the same decidability property for 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.

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

Since carrying out this work, we have very recently become aware of an independent paper by Boneva and Talbot [1]. They present an encoding of two-counter machines (a Turing-complete formalism) into pure Mobile Ambients without restriction and without the open capability. The fragment of AC we consider in Theorem 3.6 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.

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 [6] and variants thereof. We let $P, Q, \ldots$ range over processes 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" process language which contains all the *operators* which we shall consider.

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

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 process $n[\,P\,]$ is an ambient named n containing process P. The process $P \mid Q$ is the parallel composition of P and Q. The process $M.P$ performs capability M and then continues with P. The process $\nu n\, P$ is process P with name n restricted. As usual, restriction is a variable-binding operator. We denote the set of free names of a process P by $\mathsf{fn}(P)$. The process $!\,P$ is a replicated process which can spin off copies of P as required. The process $\mathsf{rec}\, X.P$ is a recursion in which X is a bound process variable. We shall only consider processes where all process variables are bound. Recursion is *unboxed* [21,4] 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:

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

The first capability $\mathsf{open}\, n$ is used to dissolve an ambient named n. 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. Sometimes we consider the "safe" versions [13] 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 [20]. 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.

Structural congruence $\equiv$ equates processes which are the same up to structural rearrangement. It is defined by the following rules:

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

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.

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

We shall be considering languages which only possess a subset of the full set of capabilities. When we consider languages with capability $\overline{\mathsf{in}}$, we shall always have capability in as well, and we shall adopt rule (SafeIn) and not rule (In). Clearly, if a language has capabilities in, $\overline{\mathsf{in}}$ and replication on these capabilities, then the effect of rule (In) can be simulated; every ambient can be made perfectly receptive to entering processes by converting $n[\, P\,]$ into $n[\, !\, \overline{\mathsf{in}}\, n \mid P\,]$. Similar considerations apply to capabilities $\overline{\mathsf{out}}$ and out.

The remaining rules for reduction are

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

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

A *language* is a pair $(L, \rightarrow)$ consisting of a set of processes L together with a reduction relation $\rightarrow$. We shall write $(L, \rightarrow)$ 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 $P_0 \rightarrow P_1 \rightarrow \cdots$.

The most basic observation that can be made of a process is the presence of top-level ambients (i.e. ambients which are not guarded by capabilities or contained in other ambients) [6]. We say that n is a *strong barb* of P ($P \downarrow n$) iff $P \equiv \nu m_1, \ldots m_k\, (n[\,Q\,] \mid R)$ (where $n \neq m_1, \ldots m_k$), and n is a *weak barb* of P ($P \Downarrow n$) iff $P \Rightarrow P' \downarrow n$.

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 [6] established that pure Mobile Ambients can encode Turing machines. Busi and Zavattaro [4] improved this result by showing that counter machines (CMs) can be encoded in pure MA without restriction. They also showed that the fragment of pure MA with no movement capabilities (but with restriction) can encode CMs.

We shall show that CMs can be encoded in pure MA without restriction and without open. 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 [22]. They use a set of instructions which is minimal while retaining Turing completeness [18]. (In fact CMs with just 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. 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 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.*

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 $P \Downarrow n$.

Furthermore, our encodings will actually satisfy both Criterion 3.1 and the following additional property:

Criterion 3.2 • *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 3.2 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. Nevertheless, one can achieve separation results by showing Criterion 3.2 for one fragment and decidability of termination for another fragment.

Many encodings 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.

Remark 3.3 Hirschkoff, Lozes and Sangiorgi [11] give an encoding of TMs into a fragment of AC which satisfies one-step preservation, but 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 termination. This is sufficient for them to claim Turing completeness, but we shall require that computations cannot take unintended paths.

3.2 Existing Work

Busi and Zavattaro gave encodings of CMs into two fragments of the pure AC. The first fragment, which we shall call L_{ν}^{op}, is defined by

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

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. Nevertheless, this result shows the strength of the open capability. We wish to investigate whether we can achieve Turing completeness without open.

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. Independently, Hirschkoff, Lozes and Sangiorgi [11] 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.

3.3 "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 3.4) and for subjective movement (Theorem 3.5).

Let $L_{\mathsf{ppa}}^{\mathsf{op}}$ be the following language (a fragment of the Push and Pull Ambient Calculus [20]):

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

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

Theorem 3.4 *$L_{\mathsf{ppa}}^{\mathsf{op}}$ is Turing complete.*

Proof. (Sketch) We describe an encoding of CMs into $L_{\mathsf{ppa}}^{\mathsf{op}}$. 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 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

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

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 by $[\![I_i]\!]_l$, defined as follows:

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

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

$$S_{ij} \overset{\text{df}}{=} d_i[\,\mathsf{pull}\ s \mid r_j[\,\mathsf{pull}\ s \mid \mathsf{open}\ s.(e_i[\,] \mid \mathsf{push}\ e_i)\,] \mid \mathsf{push}\ e_i \mid \mathsf{st}_{i+1}[\,]\,] \mid \\ \mathsf{open}\ e_i.\mathsf{push}\ d_i$$

$$Z_{iji'} \overset{\text{df}}{=} \mathsf{open}\ z.(d'_i[\,r_j[\,\underline{0}\,] \mid \mathsf{st}_{i'}[\,]\,] \mid \mathsf{push}\ d'_i)$$

We use P^k to abbreviate k copies of P in parallel. Notice that the continuations of all occurrences of open are finite (the same condition as used in [11] and mentioned in the previous subsection).

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:

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

  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:

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

  Again, computation is entirely deterministic.

Finally, we see that if CM_L is terminal (so $i_L = a + 1$) 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 3.5 $L^{\mathsf{op}}_{\mathsf{ioa}}$ *is Turing complete. (Proof: see Appendix A.)*

3.4 Languages without open

So far, all the languages considered have possessed the open capability. We shall show that this is not essential for Turing completeness, by encoding 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 3.6 L_{io} *is Turing complete.*

Proof. (Sketch) 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's still present. This issue does not arise if we can 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 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 nonterminating 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 3.4, 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.

See Appendix B for the details. □

Remark 3.7 In independent work, Boneva and Talbot [1] 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:

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

Theorem 3.8 *L_{pp} is Turing complete. (Proof: see Appendix C.)*

4 Terminating Fragments of AC

We would like to know whether the language L_{io} of Subsection 3.4 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 $(L, \rightarrow)$ is *terminating* if every computation is finite.

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

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

Notice that $L_{\mathsf{i}\overline{\mathsf{i}}}$ is got from L_{io} by removing the out capability and (in order to sharpen the next theorem) adding the co-capability $\overline{\mathsf{in}}$ [13] and restriction.

Theorem 4.2 *$L_{\mathsf{i}\overline{\mathsf{i}}}$ is terminating.*

Proof. (Sketch) First observe that if a process P of $L_{\mathsf{i}\overline{\mathsf{i}}}$ has an infinite computation, then if we identify all the names of P and remove all restrictions, we still have an infinite computation, as all existing reductions can still occur (as well as potentially some new ones). Similarly, we can replace all capabilities by replicated capabilities. Thus it suffices to show the theorem for the sublanguage

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

where m is a single fixed name.

We first define the *replication nesting depth (rnd)* of a process:

$$\mathsf{rnd}(\mathbf{0}) \overset{\text{df}}{=} 0 \qquad \mathsf{rnd}(P \mid Q) \overset{\text{df}}{=} \max(\mathsf{rnd}(P), \mathsf{rnd}(Q))$$

$$\mathsf{rnd}(m[\,P\,]) \overset{\text{df}}{=} \mathsf{rnd}(P) \qquad \mathsf{rnd}(!\,\mathsf{in}\, m.P) \overset{\text{df}}{=} \mathsf{rnd}(P) + 1$$

$$\mathsf{rnd}(!\,\overline{\mathsf{in}}\, m.P) \overset{\text{df}}{=} \mathsf{rnd}(P) + 1$$

We next define the *replication degree* (abbreviated to rd, or simply degree) of an ambient $m[\,P\,]$. This is the rnd of the *capability component* of P, defined as follows. Any process P is structurally congruent to

$$\prod_{1 \le i \le I} !\,\mathsf{in}\, m.P_i \mid \prod_{1 \le j \le J} !\,\overline{\mathsf{in}}\, m.P_j \mid \prod_{1 \le k \le K} m[\,P_k\,]$$

where $I, J, K \ge 0$, and $I = 0$ indicates that the parallel composition is empty (similarly for J, K). The capability component of P is

$$P^{cap} \overset{\text{df}}{=} \prod_{1 \le i \le I} !\,\mathsf{in}\, m.P_i \mid \prod_{1 \le j \le J} !\,\overline{\mathsf{in}}\, m.P_j$$

and we let $\mathsf{rd}(m[\,P\,]) \overset{\text{df}}{=} \mathsf{rnd}(P^{cap})$. This is well-defined with respect to structural congruence. Notice that the degree of an ambient is unchanged throughout a computation. It is unaffected by other ambients entering of whatever degree. Also, no capability can ever disappear.

During a computation an ambient can produce "children". For instance $m[\,!\,\mathsf{in}\, m.m[\,]\,]$ can produce a series of new $m[\,]$ ambients as it enters other ambients. These children will have strictly lower replication 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 reaction.

We can assume that all ambients are equipped with both $!\,\mathsf{in}$ and $!\,\overline{\mathsf{in}}$ capabilities. Thus all ambients have degree ≥ 1.

We sketch 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 $P_0 \to \cdots$ 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 degree less than their parents, and $< D_0$. Since the computation is infinite, infinitely many children must be created. 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 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 degree $> D$.

We have shown how to assign a pair (D, c) ($D \geq 1$, $c \geq 1$) to each infinite computation. Now let $P_0 \to \cdots \to P_i \to \ldots$ be an infinite computation with a minimal value of (D, c) in the 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 of (D, c).

Choose any infinitely productive ambient of degree $> D$. We can assume that it is available at the start of the computation, by removing a finite initial segment of the computation if necessary (this does not change the values of D and c). Each process P_i of the computation is of the form $\mathcal{C}_i\{m[\,Q_i\,]\}$, where we display the outer context and inner contents of our chosen ambient. Reductions either involve the context alone, the contents alone, or else they involve the chosen ambient as a *principal*—either entering or being entered. Since the ambient is infinitely productive, there must be infinitely many of this third type of reduction.

Now let us alter the computation by making the chosen ambient totally unproductive—simply remove the continuations of the capabilities exercised by the ambient. We still have an infinite computation, which is less productive than before. If the value of D is not reduced, then the value of c must have been reduced by at least one. Hence our new computation is lower in the lexicographic ordering, which is a contradiction.

Method 2. Let $\to'$ be the modification of standard $\to$ reduction (Section 2) where reduction is forbidden inside an ambient. Let $\Rightarrow'$ be the reflexive and transitive closure of $\to'$.

Suppose there is an infinite computation starting from

$$P \equiv P^{cap} \mid \prod_{1 \leq k \leq K} m[\,P_k\,]$$

Then we must have $K \geq 1$ for P to have a reduction. If $K = 1$ we write $P \searrow P_1$. Clearly P_1 has an infinite computation. If $K > 1$, one can show that there is an infinite computation of P which begins with one particular top-level ambient being entered by all the other top-level ambients. Thus $P \Rightarrow' P'$ where P' has a $\searrow$ reduction. Putting all this together, we see that if P has an infinite $\rightarrow$ computation then P has an infinite $\Rightarrow'\searrow$ computation.

To show that infinite $\Rightarrow'\searrow$ 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 $\Rightarrow'\searrow$.

Let $P_0, \ldots, P_i, \ldots$ be an infinite $\Rightarrow'\searrow$ computation. We assign to each P_i a multiset S_i. Its elements will be ordered pairs consisting of a natural number and a multiset of natural numbers. These numbers are all degrees of (unguarded) ambients in P_i.

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

In the computation there are two kinds of reductions: $\rightarrow'$ and $\searrow$. Suppose that $P_i \rightarrow' P_{i+1}$. A $\rightarrow'$ reduction consists of an ambient $m[\,Q\,]$ of degree d entering an ambient $m[\,Q'\,]$ of degree d'. To these ambients there correspond elements (d, T) and (d', T') in S_i. (Since we are doing a top-level reduction the two ambients are represented in the first elements of the pairs of S_i.) The $\rightarrow'$ reduction will produce children of $m[\,Q\,]$ of degree $< d$; we add their degrees to T. The reduction will also produce children of $m[\,Q'\,]$ of degree $< d'$; we add their degrees to T'. In this way we create S_{i+1}.

Now suppose that $P_i \searrow P_{i+1}$. A $\searrow$ reduction essentially discards a top-level ambient, while keeping its contents. Suppose this ambient is of degree d and corresponds to the element (d, T) of S_i. We remove the (d, T) from S_i and for each $d' \in T$ we add $(d', \emptyset)$ to S_i. Note that each $d' < d$. In this way we create S_{i+1}.

One can define an ordering on multisets over a well-founded ordering, by which $S \succ S'$ if S' is got from S by replacing any element of S by a finite set of smaller elements. This ordering is well-founded [8]. Now if we consider just the first members of the pairs in the multisets S_i we see that a $\rightarrow'$ reduction leaves the set unchanged, while a $\searrow$ reduction removes one element and replaces it with a finite set of smaller elements. So each $\Rightarrow'\searrow$ reduction takes us down in the $\succ$ ordering. By well-foundedness of $\succ$ there is no infinite $\Rightarrow'\searrow$ computation, and thus no infinite $\rightarrow$ computation. $\square$

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} by removing the in capability and (in order to sharpen the next theorem) adding restriction.

Theorem 4.3 *L_{o} is terminating. (Proof: see Appendix D.)*

Notice that this is not the case in the language where we add co-capability $\overline{\mathsf{out}}$ to L_{o}, 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 [16]: $n[\,n[\,\mathsf{out}\,n\,]\,] \mid !\,\overline{\mathsf{out}}\,n.n[\,n[\,\mathsf{out}\,n\,]\,]$. With "push" as the only capability we can have infinite computations, e.g. $n[\,n[\,] \mid !\,\mathsf{push}\,n.n[\,]\,]$.

Remark 4.4 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 [4] showed that termination is decidable for processes built with replication and open (see Theorem 5.2 in Section 5).

5 Fragments of AC with Decidable Termination

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.

Busi and Zavattaro showed that termination is decidable in a language without restriction, and with open but no movement capabilities. They are able to allow *unboxed* recursion rather than merely replication. Their proof relies on the facts that any process has only finitely many names (since restriction is absent), and that there is a finite bound on the nesting depth of ambients. These properties remain true if we add the out and push capabilities, since these cannot increase nesting depth of ambients.

Let $L^{\mathsf{op}}_{\mathsf{o}}$ be the following language:

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

Recursion is unboxed in $L^{\mathsf{op}}_{\mathsf{o}}$.

Theorem 5.2 *Termination is decidable for $L^{\mathsf{op}}_{\mathsf{o}}$.*

Proof. (Sketch) Straightforward adaptation of the proof of Corollary 4.10 of [4]. Busi and Zavattaro's method is to show that a multiset-style ordering on processes, under which, for instance, P is below $P \mid Q$, is a well-quasi-ordering. They then use the theory of well-structured transition systems [9] to deduce that termination is decidable. We make appropriate changes to their ordering on processes to incorporate the added capabilities. □

Remark 5.3 We know that termination is undecidable for L_{io} (see proof of Theorem 3.6). It follows from Theorem 5.2 that there can be no embedding $[\![-]\!]$ from L_{io} into $L^{\mathsf{op}}_{\mathsf{o}}$ which respects termination, in the sense that for any process P of L_{io}, P has a nonterminating computation iff $[\![P]\!]$ has a nonterminating computation.

Matters are different when it comes to the in capability and full replication (rather than replication on capabilities, as considered in Section 4). Even such a simple process as $!\,n[\,\mathsf{in}\, n\,]$ can have a computation with unbounded ambient nesting depth. The proof method of Theorem 5.2 is therefore not available.

Let L_{in} be the following language:

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

Theorem 5.4 *Termination is decidable for L_{in}.*

Proof. (Sketch) We first convert a process P of L_{in} into a process P^N by removing any occurrence of replication except on ambients and capabilities using the following equations (sometimes included in $\equiv$):

$$!\,\mathbf{0} = \mathbf{0} \quad !\,(P \mid Q) = !\,P \mid !\,Q \quad !\,!\,P = !\,P$$

One can show that P has an infinite computation iff P^N has an infinite computation. So we may now assume that L_{in} only allows replication on ambients and capabilities.

To decide whether a process P of L_{in} has a non-terminating computation, we shall translate P into a non-standard process language L^D_{in} which has a reduction relation $\rightarrow^D$ which traps non-termination finitely, so that every computation terminates.

Let L^D_{in} be the following language:

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

Here DIV represents divergence. We translate a process P of L_{in} into a process P^D of L^D_{in} using a function which is homomorphic apart from the case of a replicated ambient:

$$(!\,n[\,P\,])^D \stackrel{\text{df}}{=} n[\,P'^D\,] \mid {}^{!}[\,P\,] \tag{1}$$

Here P' is got from P by removing the capability component of P, i.e. any in or replicated in capabilities which $n[\,P\,]$ can exercise. We replace all replicated ambients after unfolding them exactly once. The replacement ${}^{!}[\,P\,]$ is not an ambient and has nonstandard reduction rules to be given shortly. The spun-off ambient $n[\,P'^D\,]$ is immobile, but available to be entered by other ambients.

Structural congruence on L^D_{in} is defined as in Section 2. The reduction relation $\rightarrow^D$ on L^D_{in} is defined as follows: We let $\rightarrow^D$ have all applicable rules defining standard reduction $\rightarrow$ in Section 2. To these rules we add the following two rules which trap divergence caused by replicated ambients:

$$\text{(InDiv)} \qquad {}^{!}[\,\mathsf{in}\, m.P \mid Q\,] \mid m[\,R\,] \rightarrow^D \mathsf{DIV}$$

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

Notice that ${}^{!}[\,P\,]$ can engage in at most one reduction, and that DIV has no reductions. Therefore we can adapt Theorem 4.2 to show that every computation in L^D_{in} terminates. Furthermore, every process has a finite computation tree which can be constructed effectively. Then a process P of L_{in} has a non-terminating computation iff there is any occurrence of DIV in the computation tree of P^D.

It is clear that any occurrence of DIV reflects an infinite computation of P. Thus if the rule (InDiv) is used in the tree of P^D, then P must be reducible to a process with a subterm having a divergent computation of the form

$$!\,n[\,\mathsf{in}\, m.P' \mid Q\,] \mid m[\,R\,] \rightarrow !\,n[\,\mathsf{in}\, m.P' \mid Q\,] \mid m[\,n[\,P' \mid Q\,] \mid R\,] \rightarrow \cdots .$$

Also, if the rule (AmbDiv) is used in the tree of P^D then P must be reducible to a process with a subterm having a divergent computation of the form $!\,n[\,P'\,] \rightarrow n[\,P''\,] \mid !\,n[\,P'\,] \rightarrow \cdots$ where $P' \rightarrow P''$.

In the other direction, we must show that we have not limited too much the scope of P^D to diverge. Equation (1) limits replicated ambients in two ways:

Firstly, the spun-off ambient is immobilised. This is justified because if the spun-off ambient were ever in a position to enter another ambient, then rule (InDiv) would also apply.

Secondly, only one ambient is spun off to create P^D, while in L_{in} we allow indefinitely many. But if an unbounded number of spun-off ambients each do at least one reduction, rule (AmbDiv) would apply. If an unbounded number of spun-off ambients get entered, then each of these entries can be simulated using the single spun-off ambient of P^D. (Of course this cannot in fact happen since L^D_{in} is terminating.) Finally we might have a divergent computation emerge in one particular spun-off copy after various ambients have entered. But if all ambients enter the same spun-off ambient, this divergence can also emerge. We conclude that P^D does indeed trap every possible divergence of P, as required. $\square$

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.

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).
- 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.
- We have found interesting links between our Method 2 in the proof of Theorem 4.2 and the proof of Theorem 2 of [12]: exploring this relation might lead to the discovery of interesting links with proof theory and independence results for Peano Arithmetic.

References

[1] I. Boneva and J.-M. Talbot. When ambients cannot be opened. In *Proceedings of FoSSaCS 2003*, volume 2620 of *Lecture Notes in Computer Science*, pages

169–184. Springer-Verlag, 2003.

[2] M. Bugliesi, G. Castagna, and S. Crafa. Boxed ambients. In *Proceedings of TACS'01*, volume 2215 of *Lecture Notes in Computer Science*, pages 38–63. Springer-Verlag, 2001.

[3] M. Bugliesi, S. Crafa, M. Merro, and V. Sassone. Communication interference in mobile boxed ambients. In *Proceedings of FSTTCS'02*, volume 2556 of *Lecture Notes in Computer Science*, pages 71–84. Springer-Verlag, 2002.

[4] N. Busi and G. Zavattaro. On the expressiveness of movement in pure mobile ambients. In *Proceedings of F-WAN: Workshop on Foundations of Wide-area Network Computing, Malaga, July 2002*, volume 66 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[5] L. Cardelli and A.D. Gordon. Mobile ambients. In *Proceedings FoSSaCS'98*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer-Verlag, 1998. Full version appears as [6].

[6] L. Cardelli and A.D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.

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

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

[9] A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256:63–92, 2001.

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

[11] D. Hirschkoff, E. Lozes, and 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*, pages 423–432. IEEE Computer Society Press, 2002.

[12] L. Kirby and J. Paris. Accessible independence results for Peano arithmetic. *Bulletin of the London Mathematical Society*, 14:285–293, 1982.

[13] F. Levi and D. Sangiorgi. Controlling interference in ambients. In *Proceedings of the 27th Annual Symposium on Principles of Programming Languages, POPL '00*. ACM, 2000.

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

[15] S. Maffeis and I.C.C. Phillips. On the computational strength of pure ambient calculi. In *Proceedings of Express: Workshop on expressiveness in concurrency, Marseille, September 2003*, volume 91 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2003.

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

[17] M. Merro and V. Sassone. Typing and subtyping mobility in boxed ambients. In *Proceedings of CONCUR'02*, volume 2421 of *Lecture Notes in Computer Science*, pages 304–320. Springer-Verlag, 2002.

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

[19] A. Phillips, S. Eisenbach, and D. Lister. From process algebra to Java code. In *Proceedings of the ECOOP workshop on Formal Techniques for Java-like programs, Malaga, Spain*, 2002.

[20] I.C.C. Phillips and M.G. 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.

[21] D. Sangiorgi. Extensionality and intensionality of the ambient logics. In *Proceedings of POPL'01*, pages 4–17. ACM, 2001.

[22] J.C. Shepherdson and J.E. Sturgis. Computability of recursive functions. *Journal of the Association for Computing Machinery*, 10:217–255, 1963.

[23] D. Teller, P. Zimmer, and D. Hirschkoff. Using ambients to control resources. In *Proceedings of CONCUR 2002*, volume 2421 of *Lecture Notes in Computer Science*, pages 288–303. Springer-Verlag, 2002.

[24] P. Zimmer. On the expressiveness of pure mobile ambients. In *Proceedings of Express'00*, volume 39 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2001.

Appendix

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 Subsection 3.3:

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

Theorem 3.5. *$L^{\mathsf{op}}_{\mathsf{ioa}}$ is Turing complete.*

Proof. (Sketch) The proof of Turing equivalence follows the structure of the one of Theorem 3.4.

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

$$\underline{0} \overset{\mathrm{df}}{=} z[\,\mathsf{in}\, jz\,]$$

$$\underline{k+1} \overset{\mathrm{df}}{=} s[\,\underline{k} \mid \mathsf{in}\, ds\,]$$

$$[\![r_j(k)]\!]_l \overset{\mathrm{df}}{=} r_j[\,\mathsf{in}\, r_j \mid \underline{k}\,]$$

The encoding is completely deterministic, since at each step only one reduction is possible. The initial state is defined as

$$\Pi_i\ [\![I_i]\!]_0 \mid \Pi_j\ [\![r_j]\!]_0 \mid \mathsf{st}_0[\ ]$$

To increment a register r_j, we first make it enter in a dummy copy of itself which, once acknowledges the presence of the register, moves in 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}.

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

The instruction for decrement and 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.

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

The strategy consists in opening the instruction trigger, 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).

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

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

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

after the first three steps we reach

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

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

$$\ldots \mid r_j[\,ds[\,\underline{n} \mid \mathsf{in}\ dds_i \mid \mathsf{in}\ b\,] \mid djz_i[\,b[\,\mathsf{open}\ jz.c[\,\mathsf{out}\ b\,]\,] \mid \mathsf{open}\ c.\mathsf{out}\ r_j\,] \mid \ldots\,] \mid \ldots$$

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.

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

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

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

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

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

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

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

□

B Encoding of CMs into L_{io}

We present an encoding of CMs into the language L_{io} defined in Subsection 3.4:

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

Theorem 3.6. *L_{io} is Turing complete.*

Proof. (Sketch) 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's still present. This issue does not arise if we can 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 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. 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 to report the result.

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 register and instructions, we must take into account the fact that the 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
- $dec(i, l)$ be the number of decrements
- $dec_s(i, l)$ 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. Clearly, $dec(i, l) = dec_s(i, l) + dec_z(i, l)$.

$$\begin{aligned} [\![R_0(k)]\!]_l &\overset{\text{df}}{=} r_0[\,\underline{k}_l \mid \textstyle\prod_{i \leq a+2} !\,\mathsf{in}\ \mathsf{st}_i\,] \\ [\![R_j(k)]\!]_l &\overset{\text{df}}{=} r_j[\,\underline{k}_l \mid \textstyle\prod_{i \leq a} !\,\mathsf{in}\ \mathsf{st}_i\,] \quad (1 \leq j \leq b) \end{aligned}$$

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:

$$\begin{aligned} \underline{0}_l &\overset{\text{df}}{=} z[\,IZ \mid D_t \mid (increq[\,!\,\mathsf{in}\ s.\mathsf{in}\ t\,])^{inc(i,l)}\,] \\ IZ &\overset{\text{df}}{=} !\,\mathsf{in}\ s.\mathsf{in}\ t \\ D_t &\overset{\text{df}}{=} !\,\mathsf{in}\ dect'.\mathsf{out}\ dect'.\mathsf{out}\ t.\mathsf{out}\ dect \end{aligned}$$

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.

$$\begin{aligned} \underline{k+1}_l &\overset{\text{df}}{=} s[\,DS \mid D_t \mid t[\,DT \mid D_s \mid \underline{k}_l\,]\,] \\ DS &\overset{\text{df}}{=} \mathsf{in}\ decs \\ DT &\overset{\text{df}}{=} \mathsf{in}\ dect \\ D_s &\overset{\text{df}}{=} \mathsf{in}\ decs'.\mathsf{out}\ decs'.\mathsf{out}\ s.\mathsf{out}\ decs \end{aligned}$$

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.

$$Mon \overset{\mathrm{df}}{=} m[\prod_{i\le a} !\,\mathsf{in}\ \mathsf{st}_i.M_i\,]$$

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

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 $(i \le a)$ is of the form

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

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:

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

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 $i \le a$, the st_i ambient does nothing further once it has appeared at the current level; it is convenient to define $F_i \overset{\mathrm{df}}{=} \mathbf{0}$ $(i \le a)$.

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:

$$\mathcal{C}_0\{\bullet\} \overset{\mathrm{df}}{=} \bullet$$

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

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

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

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.

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

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

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:

$$\begin{aligned} P_i &\overset{\text{df}}{=} \mathit{test}[\,\mathsf{out}\,p_i.\mathsf{in}\,r_j.(Q_z \mid Q_s)\,] \\ Q_z &\overset{\text{df}}{=} \mathsf{in}\,z.\mathsf{out}\,z.\mathsf{out}\,r_j.\mathsf{in}\,p_i.\mathsf{st}_{i'}[\,\mathsf{out}\,\mathit{test}.\mathsf{out}\,p_i.(\,!\,\mathsf{out}\,t.\mathsf{out}\,s \mid F_{i'})\,] \\ Q_s &\overset{\text{df}}{=} \mathsf{in}\,s.\mathsf{out}\,s.\mathsf{out}\,r_j.\mathsf{in}\,p_i.P'_i \end{aligned}$$

The test ambient enters r_j. If it detects z it leaves the register, re-enters p_i and unleashes instruction i'. The process $\mathit{test}[\,Q_s\,]$ 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 $\mathit{test}[\,Q_z\,]$ remains as garbage inside p_i. (The $!\,\mathsf{out}\,t.\mathsf{out}\,s$ inside Q_z is not used, and is simply included for uniformity with the increment case described above, where it is needed.)

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

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

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:

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

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.

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

This is detected by decs, which exits r_j, enters p_i and unleashes P''_i. The first stage is completed. The process $\mathit{decs}[\,s[\,D_t \mid \mathit{decs}'[\,]\,]\,]$ 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.

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

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

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

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 nonterminating 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 a couple of CM instructions, which it is convenient to number $a+1$, $a+2$. Thus the next CM starts its instructions proper at $a+3$. We define its monitor process in such a way that the old R_0 is not expected to travel beyond instruction $a+2$. 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 Subsection 3.4:

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

Theorem 3.8. *L_{pp} is Turing complete.*

Proof. (Sketch) 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 ($j \le b$). 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 register and instructions, we must take into account the fact that the 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:

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

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.

$$SZ_j \stackrel{\mathrm{df}}{=} \mathsf{pull}\ z_j.\mathsf{push}\ incack_j$$
$$TZ_j \stackrel{\mathrm{df}}{=} \mathsf{pull}\ z_j.(\mathsf{push}\ incack_j \mid incack_j[\,])$$

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

$$I_j \stackrel{\mathrm{df}}{=} !\,\mathsf{pull}\ increq_j.\mathsf{push}\ incack_j$$

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

$$SD_j \stackrel{\mathrm{df}}{=} \mathsf{pull}\ u_j.\mathsf{push}\ t_j$$
$$TD_j \stackrel{\mathrm{df}}{=} \mathsf{pull}\ decreq_j.(TDS_j \mid TDZ_j)$$
$$TDS_j \stackrel{\mathrm{df}}{=} \mathsf{push}\ s_j.(\mathsf{push}\ decack_j \mid decack_j[\,])$$
$$TDZ_j \stackrel{\mathrm{df}}{=} \mathsf{push}\ z_j.(\mathsf{push}\ decack_j \mid decack_j[\,])$$

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

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

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

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

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

$$!\,p_i[\,\mathsf{pull}\ \mathsf{st}_i.(test[\,Testz_j \mid Tests_j\,] \mid Dotest_{iji'})\,]$$
$$\mid (GJ_{ij})^{jump(i,l)} \mid (GDS_{iji'})^{dec_s(i,l)} \mid (GDZ_{iji'})^{dec_z(i,l)}$$

where

$$Testz_j \stackrel{\mathrm{df}}{=} \mathsf{pull}\ z_j.\mathsf{push}\ z_j.(tested[\,] \mid$$
$$\mathsf{push}\ tested.\mathsf{pull}\ back.(\mathsf{push}\ jump \mid jump[\,]))$$
$$Tests_j \stackrel{\mathrm{df}}{=} \mathsf{pull}\ s_j.\mathsf{push}\ s_j.(tested[\,] \mid$$
$$\mathsf{push}\ tested.\mathsf{pull}\ back.(\mathsf{push}\ decreq_j \mid decreq_j[\,DR_j\,]))$$
$$DR_j \stackrel{\mathrm{df}}{=} u_j[\,] \mid \mathsf{pull}\ s_j.\mathsf{push}\ t_j$$
$$Dotest_{iji'} \stackrel{\mathrm{df}}{=} \mathsf{push}\ test.\mathsf{pull}\ tested.\mathsf{pull}\ test.(back[\,] \mid FJ_{i'} \mid FD_{ij})$$
$$FJ_{i'} \stackrel{\mathrm{df}}{=} \mathsf{push}\ jump.\mathsf{pull}\ jump.(\mathsf{push}\ \mathsf{st}_{i'} \mid \mathsf{st}_{i'}[\,])$$
$$FD_{ij} \stackrel{\mathrm{df}}{=} \mathsf{push}\ decreq_j.\mathsf{pull}\ decack_j.\mathsf{pull}\ t_j.(\mathsf{push}\ \mathsf{st}_{i+1} \mid \mathsf{st}_{i+1}[\,])$$

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):

$$\begin{aligned}GJ_{ij} &\overset{\mathrm{df}}{=} p_i[\,\mathsf{st}_i[\,] \mid \mathit{tested}[\,] \mid FD_{ij} \mid \mathit{test}[\,\mathit{back}[\,] \mid \mathit{Tests}_j\,] \\ &\quad \mid \mathit{jump}[\,]\,] \\ GDZ_{iji'} &\overset{\mathrm{df}}{=} p_i[\,\mathsf{st}_i[\,] \mid \mathit{tested}[\,] \mid FJ_{i'} \mid \mathit{test}[\,\mathit{back}[\,] \mid \mathit{Testz}_j\,] \\ &\quad \mid \mathit{decack}_j[\,] \mid t_j[\,\mathit{decreq}_j[\,s_j[\,u_j[\,] \mid I_j\,]\,] \mid TDS_j \mid I_j\,]\,] \\ GDS_{iji'} &\overset{\mathrm{df}}{=} p_i[\,\mathsf{st}_i[\,] \mid \mathit{tested}[\,] \mid FJ_{i'} \mid \mathit{test}[\,\mathit{back}[\,] \mid \mathit{Testz}_j\,] \\ &\quad \mid \mathit{decack}_j[\,] \mid t_j[\,\mathit{decreq}_j[\,s_j[\,u_j[\,] \mid I_j\,]\,] \mid TDZ_j \mid I_j\,]\,]\end{aligned}$$

We define:

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

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 $[\![CM_{l+1}]\!]_{l+1}$. Computation is entirely deterministic. There are various cases, depending on the kind of instruction.

First consider the execution of $[\![i:\mathsf{Inc}(j))]\!]_l$. Starting from

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

the instruction is activated (ambient p_i), and the $\mathit{increq}_j[\,]$ ambient is pushed to the top level:

$$[\![i:\mathsf{Inc}(j)]\!]_l \mid p_i[\,\mathsf{st}_i[\,] \mid \mathsf{pull}\,\mathit{incack}_j.(\ldots)\,] \mid \mathit{increq}_j[\,] \mid [\![R_j(k)]\!]_l$$

Then the $\mathit{increq}_j[\,]$ 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.

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

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

$$\ldots s_j[\,\mathsf{push}\,\mathit{incack}_j \mid SD_j \mid I_j \mid t_j[\,\mathsf{push}\,\mathit{incack}_j \mid \mathit{incack}_j[\,] \mid I_j \mid z_j[\ldots]\,]\,]\ldots$$

The acknowledgement $\mathit{incack}_j[\,]$ 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 used instruction $p_i[\,\mathsf{st}_i[\,] \mid \mathit{incack}_j[\,]\,]$ (i.e. GI_{ij}) is now at the top level, 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.

$$\ldots p_i[\,\mathsf{st}_i[\,] \mid \mathsf{pull}\,\mathit{tested}.(\ldots)\,] \mid \mathit{test}[\,\mathit{Testz}_j \mid \mathit{Tests}_j\,] \mid [\![R_j(k)]\!]_l$$

Once it has done the test it signals this to p_i with ambient *tested*. Both ambients are then pulled back into p_i. Ambient *tested* discovers that it is back in p_i by pulling in ambient *back*. There are now two possibilities, depending on whether k is zero or nonzero.

1. k is zero.

$$p_i[\,\mathsf{st}_i[\,] \mid tested[\,] \mid FJ_i \mid FD_{ij} \mid$$
$$test[\,back[\,] \mid \mathsf{push}\, jump \mid jump[\,] \mid Tests_j\,]\,] \ldots$$

Then *test* pushes out *jump*, which is detected by p_i, which 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

$$p_i[\,\mathsf{st}_i[\,] \mid tested[\,] \mid FD_{ij} \mid test[\,back[\,] \mid Tests_j\,] \mid jump[\,]\,]$$

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

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

2. k is nonzero.

$$p_i[\,\mathsf{st}_i[\,] \mid tested[\,] \mid FJ_{i'} \mid FD_{ij} \mid$$
$$test[\,back[\,] \mid \mathsf{push}\, decreq_j \mid decreq_j[\,DR_j\,] \mid Testz_j\,]\,] \ldots$$

Then *test* and p_i push out ambient $decreq_j$ to carry out the decrement. Then $decreq_j$ pulls in s_j (the entire register).

$$p_i[\,\mathsf{st}_i[\,] \mid tested[\,] \mid FJ_{i'} \mid \mathsf{pull}\, decack_j.(\ldots) \mid test[\,back[\,] \mid Testz_j\,]\,] \mid$$
$$decreq_j[\,u_j[\,] \mid \mathsf{push}\, t_j \mid s_j[\,SD_j \mid I_j \mid t_j[\,TD_j \mid I_j \mid [\![R_j(k-1)]\!]_l\,]\,]\,]$$

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 $decreq_j$.

$$p_i[\,\mathsf{st}_i[\,] \mid tested[\,] \mid FJ_{i'} \mid \mathsf{pull}\, decack_j.(\ldots) \mid test[\,back[\,] \mid Testz_j\,]\,] \mid$$
$$t_j[\,decreq_j[\,s_j[\,u_j[\,] \mid I_j\,]\,] \mid TDS_j \mid TDZ_j \mid I_j \mid [\![R_j(k-1)]\!]_l\,]$$

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 $k - 1 > 0$:

$$p_i[\,\mathsf{st}_i[\,] \mid tested[\,] \mid FJ_{i'} \mid \mathsf{pull}\, decack_j.(\ldots) \mid test[\,back[\,] \mid Testz_j\,]\,] \mid$$
$$decack_j[\,] \mid t_j[\,decreq_j[\,s_j[\,u_j[\,] \mid I_j\,]\,] \mid TDZ_j \mid I_j\,] \mid [\![R_j(k-1)]\!]_l$$

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 $GDS_{iji'}$ or $GDZ_{iji'}$. We are left with

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

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. □

Remark C.1 Instead of letting the garbage accumulate as the computation proceeds, as above, the recent work of Boneva and Talbot [1] shows how one could introduce replicated ambients as garbage collectors.

D $L_\circ$ is terminating

Let $L_\circ$ be the following language:

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

Theorem 4.3. *$L_\circ$ is terminating.*

Proof. (Sketch) As with Theorem 4.2, it suffices to show the theorem for the sublanguage without restriction.

The idea of the proof is as follows: 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 $\succ$ described above. Each element in the multiset measures the number of ambients working from an occurrence of $\mathbf{0}$ outwards.

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

We note that if $\mathsf{ms}(P) \succ \mathsf{ms}(Q)$ then $\mathsf{ms}(\mathcal{C}\{P\}) \succ \mathsf{ms}(\mathcal{C}\{Q\})$.

We need to make some adjustments to structural congruence and reduction as defined in Section 2. Denote the new *ad hoc* reduction relation and structural congruence by $\rightarrow'$, $\equiv'$ respectively. We allow parallel composition to be commutative and associative for $\equiv'$, but we disallow

$${!\,\mathsf{out}}\, n.P \equiv' \mathsf{out}\, n.P \mid {!\,\mathsf{out}}\, n.P$$

so that replication only unfolds as part of a reaction (otherwise structurally congruent terms could have different multisets). We also disallow $P \mid \mathbf{0} \equiv' P$ for the same reason. To compensate, in addition to standard rule (Out), we adopt rule (RepOut) which unfolds a replication precisely when needed for a reduction:

$$\begin{array}{ll} \text{(Out)} & m[\,n[\,\mathsf{out}\, m.P \mid Q\,] \mid R\,] \rightarrow' n[\,P \mid Q\,] \mid m[\,R\,] \\ \text{(RepOut)} & m[\,n[\,{!\,\mathsf{out}}\, m.P \mid Q\,] \mid R\,] \rightarrow' n[\,P \mid {!\,\mathsf{out}}\, m.P \mid Q\,] \mid m[\,R\,] \end{array}$$

None of these changes removes any computation, though the terms may change in appearance. The relationship between standard $\rightarrow$ and the *ad hoc* $\rightarrow'$ is as follows: If $P \rightarrow' P'$ then $P \rightarrow P'$. Conversely, if $P \rightarrow P'$ then there exists P'' such that $P \rightarrow' P''$ and $P'' \equiv P'$. It is therefore sufficient to show that $\rightarrow'$ is terminating.

Suppose that $P_0 \rightarrow' P_0'$. We must show that $\mathsf{ms}(P_0) \succ \mathsf{ms}(P_0')$. We have two reactions to consider: In the case of (Out), $\mathsf{ms}(P_0')$ has the same size as $\mathsf{ms}(P_0)$, but some elements are reduced by one because the exiting ambient has reduced its depth. Hence $\mathsf{ms}(P_0) \succ \mathsf{ms}(P_0')$. In the case of (RepOut), we replace the elements $\{k+2 : k \in \mathsf{ms}(P)\}$ by $\{k+1 : k \in \mathsf{ms}(P) \cup \mathsf{ms}(P)\}$. Hence also $\mathsf{ms}(P_0) \succ \mathsf{ms}(P_0')$. By the well-foundedness of $\succ$, we conclude that every computation must terminate. $\square$