On the Expressiveness of Asynchronous Processes with Nested Session Types

Sjors Wortelboer Dan Frumin Jorge A. Pérez
University of Groningen, The Netherlands

Session types are a type-based approach to correct message-passing programs, where types specify protocols for channels and type checking guarantees communication correctness. Traditional session types express regular protocols only, which leaves out many protocols found in practice. Aiming at overcoming this limitation, Nested Session Types (NSTs) can specify nested recursive protocols. NSTs are known to be characterised by one-counter machines and deterministic context-free languages. In this paper, we study the expressiveness of ASP (Asynchronous Session Processes), the process model associated with NSTs as introduced by Das *et al.*. We prove that ASP is Turing complete by giving a correct encoding of two-counter machines into typed processes. Our encoding uses a non-trivial *state-aware* representation of counters and instructions that is neatly governed by NSTs.

1 Introduction

Ensuring the correctness of the message-passing programs that underpin large-scale distributed systems is a fascinating and pressing challenge. We are interested in compositional approaches to program verification where *types* specify protocols for channels and *type checking* statically guarantees communication correctness for implementations, specified as *processes* in some suitable formalism. In this approach, *expressiveness* is a primary concern, in two ways: first, we aim for rich type systems that can express communication structures found in real-world scenarios; second, since types soundly enforce correctness by constraining admissible program behaviours, we would like to determine the expressiveness of the resulting typed processes—which correct programs can be expressed in a typed setting and which cannot. In this paper, we study the expressiveness of a process language with Nested Session Types (NSTs) by Das *et al.* [9, 10]. NSTs are among the most expressive type systems for concurrency proposed to date.

Session types specify the order and type of messages to be exchanged on a channel [12, 13]. Traditional session types can express *regular* protocols, enabled by the composition of a message (input or output) and a protocol. However, many practical protocols are not regular; examples include protocols for the serialisation of tree-structured data and for buffered message relaying. The quest for more expressive session types led to the *context-free* session types by Thiemann and Vasconcelos [21]. Context-free session types support a generalised notion of protocol composition which, combined with polymorphic recursion, enables the specification of context-free protocols. NSTs achieve high expressivity using different means, namely *nested type constructors*, which allow recursive type parameters to act like a stack.

A crucial step in laying the foundations of non-regular session types is studying what we may refer to as their *intrinsic expressiveness*: the properties of these infinite structures in themselves, independent of a target programming model. Important notions for verification (type duality, type equality, subtyping) concern types in isolation. In this respect, the theory of context-free session types is well-developed [1, 2, 20]; for NSTs, Das *et al.* prove that type equality is decidable and give a sound algorithm. Gay *et al.* [11] offer a unified treatment of intrinsic expressiveness results for recursive session types. They complement the results in [9, 10] and obtain a strict hierarchy where NSTs are precisely characterised by one-counter machines and deterministic context-free languages, with decidable type equivalence and duality.

Preliminary Report. Final version to appear in:
EXPRESS/SOS 2026

In this paper, we pursue a different yet complementary goal: establishing the expressiveness of the language underlying NSTs, as introduced by Das *et al.* [9, 10]. We refer to this language as ASP (Asynchronous Session Processes). The language implements asynchronous communication in configurations, which contain processes; its formulation follows the Curry-Howard correspondence between intuitionistic linear logic and session types [8]. As a result, well-typed configurations in ASP enjoy confluence and deadlock freedom. However, the question of the expressiveness of well-typed ASP configurations remains open. Given the context-free nature of NSTs and the decidability of their key properties, we investigate whether the synchronisation mechanisms in ASP, together with the intrinsic expressiveness of types, suffice to go beyond the Chomsky hierarchy and obtain Turing completeness.

Our main result is a proof that ASP is Turing complete, which we show by encoding Minsky machines. Minsky machines [18] are a Turing-complete model that consists of two counters with a finite set of instructions; each instruction can either increment a counter, decrement it (if the counter's current value is not zero), or jump to a specified instruction (if the value is zero). With this minimal state and a small set of possible instructions, Minsky machines are, arguably, a simpler source for encoding, compared to Turing machines. Let us briefly discuss how we model Minsky machines in ASP.

Intuitively, counters can be implemented as stacks using polymorphic nested types to capture state at the type level. Given a natural m, we define the nested type

$$(\!|m|\!) = \begin{cases} 0[] & \text{if } m = 0 \\ S[(\!|m-1|\!)] & \text{if } m > 0 \end{cases}$$

where 0 and S stand for *(parametric) type definitions* for zero and successor, given as follows:

$$0[] \triangleq \&\{\texttt{incr}: S[0[]], \texttt{jump}: 0[]\} \qquad S[\alpha] \triangleq \&\{\texttt{incr}: S[S[\alpha]], \texttt{decr}: \alpha\}$$

Expressed as a branching type (&, external choice), this nested session type captures the protocol that specifies the correct state after executing an instruction, using nested types and polymorphic recursion. This way, e.g., $0[]$ admits increment and jump (but not decrement) and $S[\alpha]$ admits increment and decrement (but not jump) by suitably modifying the nesting of the type variable α, the parameter of S. Such counter types can then be combined on the process level (by a process that we will denote $[\![m]\!]_r$) with the consequence that processes representing instructions are required to be *state-aware*: they need a separate mechanism to keep track of the combined state of the two counters. This is where NSTs enable a continuation of the state-aware processes, allowing transitions to be captured at the level of processes.

Overall, we believe that the encoding of Minsky machines is a significant test of the expressiveness of the coupling of ASP and NSTs. Indeed, we achieve a pleasant design in which NSTs elegantly govern the correct behaviours of the encodings of counters and instructions, and their interactions.

The rest of the paper is structured as follows. Section 2 recalls key background notions of ASP and NSTs (following [9, 10]) and the definition of Minsky machines. The encoding of Minsky machines in ASP is presented and illustrated in Section 3; its correctness is given in terms of typability (Theorem 4) and operational correspondence (Theorems 5 and 6). Section 4 concludes the paper and discusses some related work. The appendix provides additional technical material.

2 Background

2.1 Nested Session Types

We first present Nested Session Types (NSTs), then their associated process language, which we call Asynchronous Session Processes (ASP), and finally we state the correctness properties ensured by typing.

Nested Session Types NSTs extend the Curry-Howard correspondence between session types and intuitionistic linear logic [8] with type variables and type definitions. There is a finite signature Σ that collects these type definitions. The syntax and operational interpretation of types is given next.

Definition 1 (Nested Session Types, NSTs)**.** The language of nested session types (NSTs) is defined by:

$$
\begin{array}{rll}
A,B,C ::= & A \otimes B & \text{send channel of type } A\text{, continue as } B \\
\mid & A \multimap B & \text{receive channel of type } A\text{, continue as } B \\
\mid & \oplus\{\mathtt{l} : A_l\}_{l\in L} & \text{send label } k \in L\text{, continue as } A_k \\
\mid & \&\{\mathtt{l} : A_l\}_{l\in L} & \text{receive label } k \in L\text{, continue as } A_k \\
\mid & \mathbf{1} & \text{close channel} \\
\mid & \alpha & \text{type variable} \\
\mid & V[\theta] & \text{defined type name (in } \Sigma\text{)}
\end{array}
$$

Type variables (denoted $\alpha, \beta, \ldots$) enable the parametrisation of types. NSTs do not make use of type variable binders; instead, type definitions act as binders for a sequence of type variables by which they are parameterised. Type names are defined according to a type definition, denoted $V[\overline{\alpha}] \triangleq A$, where $\overline{\alpha}$ is a list of distinct variables A can refer to. These definitions are stored in the signature Σ and allow nested type construction. The type $V[\theta]$ denotes the instantiation of a type name V, which equirecursively unfolds into A with type parameters substituted using $\theta = \overline{A}/\overline{\alpha}$. Any type that is not a type name is structural (e.g. $V_1[\theta_1] \otimes V_2[\theta_2]$). Any type that is structural and not a type variable is contractive in the type definitions. Types without free type variables are called closed types.

A key result in [10] is that type equality is decidable and has a sound algorithm. Type equality follows the judgment

$$\psi; \mathcal{V} \vdash_\Sigma A \equiv A'$$

where $\mathcal{V}$ denotes the free variables in A and A', and ψ is a collection of *closures* of the form $V[\theta] \equiv U[\sigma]$.

The Process Language We now introduce the process language ASP. This is exactly the language introduced by Das *et al.* [9, 10]. It is based on the intuitionistic variant of the Curry-Howard correspondence, under an asynchronous communication discipline.

Definition 2 (Processes and Messages)**.** Given a set of channel/names ranged over by $x, y, z, \ldots$, the syntax of *processes* and is defined by:

$$
\begin{array}{rlrl}
P,Q ::= & \mathtt{send}\ x\ y; P & \text{send channel } y \text{ over } x & \mid y \leftarrow \mathtt{recv}\ x; P \quad \text{receive a channel over } x \\
\mid & x.\mathtt{l_i};\ P & \text{send label } \mathtt{l_i} \text{ over } x & \mid \mathtt{case}\ x\ (\mathtt{l_i} \Rightarrow Q_i)_{l_i\in L} \quad \text{receive a label over } x \\
\mid & \mathtt{close}\ x & \text{closed channel } x & \mid \mathtt{wait}\ x; P \quad \text{wait for closing of } x \\
\mid & x \leftarrow f[\theta]\ \overline{y}; P & \text{process name } f[] \text{ in } \Sigma & \mid x \leftrightarrow y \quad \text{forward } x \text{ along } y
\end{array}
$$

The syntax of *messages* is defined by:

$$M ::= \mathtt{send}\ x\ y; x \leftrightarrow x' \ \mid\ x.\mathtt{k};\ x \leftrightarrow x' \ \mid\ \mathtt{close}\ x$$

The syntax of processes follows the syntax of types closely. Notable is the addition of process definitions $f[\theta]$ as a dual of nested type construction on the type level. A definition is instantiated, providing channel x to the continuation process P and making use of channels $\overline{y}$. Process definitions,

$$\frac{\mathcal{V};\Delta, y:A \vdash P :: (x:B)}{\mathcal{V};\Delta \vdash y \leftarrow \texttt{recv}\ x; P :: (x : A \multimap B)} \multimap R \qquad \frac{\cdot;\mathcal{V} \vdash A \equiv A' \qquad \mathcal{V};\Delta, x:B \vdash Q :: (z:C)}{\mathcal{V};\Delta, x:A' \multimap B, y:A \vdash \texttt{send}\ x\ y; Q :: (z:C)} \multimap L$$

$$\frac{(\forall l \in L) \qquad \mathcal{V};\Delta \vdash P_l :: (x:A_l)}{\mathcal{V};\Delta \vdash \texttt{case}\ x\ (\texttt{l} \Rightarrow P_l)_{l\in L} :: (x : \&\{\texttt{l} \Rightarrow A_l\}_{l\in L})} \&R$$

$$\frac{(k \in L) \qquad \mathcal{V};\Delta \vdash Q :: (x:A_k)}{\mathcal{V};\Delta, x:\&\{\texttt{l} \Rightarrow A_l\}_{l\in L} \vdash x.\texttt{k};\ Q :: (z:C)} \&L$$

$$\frac{\overline{y':B'} \vdash f[\overline{\alpha}] = P_f :: (x':B) \in \Sigma \qquad \cdot;\mathcal{V} \vdash \Delta' \equiv \overline{y:B'[\theta]} \qquad \mathcal{V};\Delta, x:B[\theta] \vdash Q :: (z:C)}{\mathcal{V};\Delta,\Delta' \vdash x \leftarrow f[\theta]\ \overline{y}; Q :: (z:C)} \text{def}$$

Figure 1: Selected typing rules for ASP (Definition 3).

similar to type definitions, are defined on the global signature Σ. Messages co-exist with processes as atomic processes to enable asynchronous message passing. They lift sending actions from their processes such that they can be consumed by their dual. Messages preserve linearity by renaming their channel continuation, which matches the continuation of the process producing them.

Definition 3 (Typing for Processes & Messages)**.** Given an environment of type variables $\mathcal{V}$, the typing judgment for processes and messages is denoted

$$\mathcal{V};\Delta \vdash_\Sigma P :: (x:A)$$

with selected typing rules in Figure 1 (Appendix A.1 gives a full account).

Following the intuitionistic (two-sided) presentation of linear logic, duality is defined implicitly, where a type is provided by the process over the intuitionistic channel and used by a client process over a channel in their environment. As a consequence, there are two typing rules per typing connective.

We discuss the rules. Receiving ($\multimap$) is implemented by the providing process by adding the channel y to the continuation. The dual sending action by the client process removes the channel y to preserve linearity. Furthermore, it relies on type equality on the type of channel y, such that the receiving process can use a structurally different but equivalent type. Offering a choice ($\oplus\{\cdot\}$) is implemented as branching on the providing process, where the client process selects the branch. Process definitions (def) require a well-typed definition from Σ. All types use the substitution θ where, similar to sending, dependent channels are matched using type equality to allow for better compatibility when composing processes. Finally, the continuation process simply matches the substituted type provided by the definition.

The operational semantics of the language is defined in terms of *configurations*, given next.

Definition 4 (Configurations: Syntax and Typing)**.** The syntax of configurations is defined by:

$$\mathcal{S} ::= \texttt{proc}(x,P) \mid \texttt{msg}(x,M) \mid (\mathcal{S}_1, \mathcal{S}_2)^x$$

The typing rules are as follows:

$$\frac{\cdot;\Delta \vdash P :: (x:A)}{\Delta \models_\Sigma \texttt{proc}(x,P) :: (x:A)} \text{proc} \qquad \frac{\cdot;\Delta \vdash M :: (x:A)}{\Delta \models_\Sigma \texttt{msg}(x,M) :: (x:A)} \text{msg}$$

$$\frac{\Delta_1 \models_\Sigma \mathcal{S}_1 :: (x:A) \qquad x:A, \Delta_2 \models_\Sigma \mathcal{S}_2 :: (y:B)}{\Delta_1, \Delta_2 \models_\Sigma (\mathcal{S}_1, \mathcal{S}_2)^x :: (y:B)} \text{comp}$$

$$\texttt{proc}(y, \texttt{send}\ x\ w; P) \longrightarrow \texttt{msg}(x, \texttt{send}\ x\ w; x \leftrightarrow x'),\ \texttt{proc}(y, P[x'/x]) \qquad (\multimap S)$$

$$\texttt{proc}(x, w' \leftarrow \texttt{recv}\ x; Q),\ \texttt{msg}(x, \texttt{send}\ x\ w; x \leftrightarrow x') \longrightarrow \texttt{proc}(x', Q[x'/x, w/w']) \qquad (\multimap C)$$

$$\texttt{proc}(d, x.\texttt{k};\ P) \longrightarrow \texttt{msg}(x, x.\texttt{k};\ x \leftrightarrow x'),\ \texttt{proc}(d, P[x'/x]) \qquad (\&S)$$

$$\texttt{proc}(x, \texttt{case}\ x\ (\texttt{l} \Rightarrow Q_l)),\ \texttt{msg}(x, x.\texttt{k};\ x \leftrightarrow x') \longrightarrow \texttt{proc}(x', Q_k[x'/x]) \qquad (\&C)$$

$$\texttt{proc}(z, x \leftarrow f[\theta]\ \overline{d}; Q) \longrightarrow \texttt{proc}(a, P_f[a/x, \overline{d}/\overline{y}, \theta]),\ \texttt{proc}(z, Q[a/x]) \qquad (\text{defC})$$

Figure 2: Selected reduction rules for ASP (Definition 5).

Configurations lift processes and messages with only closed types to the configuration level, as protocols need to be fully executable. The typing rule for composition binds a channel provided by some configuration $\mathcal{S}_1$ to some configuration $\mathcal{S}_2$ that uses it. This is the classical rule for composition derived from linear logic; it defines a partial ordering of the components of the configuration, and any configuration that has the same partial ordering is equivalent. We omit this presentation of composition as the bindings in the encoding are clearly defined.

We may now define reduction on the level of configurations.

Definition 5 (Reductions)**.** The reduction semantics for configurations is denoted $\longrightarrow$, with selected rules in Figure 2 (see Appendix A.2 for details).

The rules in Figure 2 illustrate the asynchronous nature of communication in ASP. Sending actions reduce independently by producing a message; a corresponding receiving action consumes the produced message separately. Consequently, a send action does not have to wait for the message to be consumed, and the receiving action does not have to be ready. Messages are produced to the right if sent over the intuitionistic channel and to the left otherwise, with renaming of channels to ensure the correct order of consumption. This preserves linearity and ensures type preservation. Similarly, process definitions produce a newly spawned process to the left of the continuation. Whenever the continuation Q is omitted, forwarding is implied as a continuation of the current process, skipping the renaming.

Properties of Typed Configurations We state here some properties of well-typed configurations. First, protocols are invariant under reduction.

Theorem 1 (Type preservation)**.** *For a well-typed configuration in* $\Delta \models_\Sigma \mathcal{S} :: (x : A)$*:*

1. *If* $\mathcal{S} \equiv \mathcal{S}'$*, then* $\Delta \models_\Sigma \mathcal{S}' :: (x : A)$*;*
2. *If* $\mathcal{S} \longrightarrow \mathcal{S}'$*, then* $\Delta \models_\Sigma \mathcal{S}' :: (x : A)$*.*

Due to the acyclic nature of composition in ASP, progress is guaranteed for internal channels. Processes receiving, or messages over external channels, are called *poised*; configurations composed of only poised components are also called poised.

Theorem 2 (Progress)**.** *For a well-typed configuration in* $\Delta \models_\Sigma \mathcal{S} :: (x : A)$*. Either* $\mathcal{S}$ *is poised, or* $\mathcal{S} \longrightarrow \mathcal{S}'$*.*

The result follows, as no cyclic dependencies that create deadlocks can occur.

Finally, confluence emerges due to the sequential independent definitions of reductions,

Theorem 3 (Confluence)**.** *Suppose a configuration $\mathcal{S}$ in ASP. If $\mathcal{S} \longrightarrow^* \mathcal{S}_1$ and $\mathcal{S} \longrightarrow^* \mathcal{S}_2$, then there exists $\mathcal{S}'$ such that $\mathcal{S}_1 \longrightarrow^* \mathcal{S}'$ and $\mathcal{S}_2 \longrightarrow^* \mathcal{S}'$.*

The language makes no choices, meaning reductions persist until consumed. Consequently, reductions can always be applied, creating different orders of application with the same resulting process.

2.2 Minsky machines

To show Turing-completeness for ASP we need an appropriate formalism. We consider *Minsky machines*, which are relatively simpler than Turing machines. Minsky machines are proven to be Turing-complete, and so any system able to simulate them is by extension Turing-complete.

Definition 6 (Minsky machines [18])**.** A Minsky machine $\mathcal{M}$ is defined as a set of instructions $\{(1 : I_1), ..., (n : I_n)\}$, where instructions are defined as:

$$\begin{array}{rll} I ::= & \texttt{INC}(j) & \text{increment on counter } j \in \{0,1\} \\ \mid & \texttt{DECJ}(j,k) & \text{decrement or jump on counter } j \in \{0,1\}, k \in \mathbb{Z}^+ \end{array}$$

A Minsky machine includes a program counter p indicating the label of the instruction being executed. The machine starts with both counters set to 0 and the program counter p set to the first instruction. The machine stops whenever the program counter is set to a non-existent instruction, i.e., $p > n$.

The *state* of a Minsky machine $\mathcal{M}$ is defined as the tuple $(i, m_0, m_1)^{\mathcal{M}}$, where $i \in \mathbb{Z}^+$ and $m_0, m_1 \in \mathbb{N}$. The tuple refers to the current instruction i and the counters with values m_0 and m_1. The state is initialised to zero values on the counters and instruction i as associated with the machine $\mathcal{M}$.

The semantics of the machine is defined as a relation on states, denoted $\longrightarrow_{\texttt{M}}$, given next:

$$\frac{i : \texttt{INC}(j) \qquad m'_j = m_j + 1 \qquad m'_{1-j} = m_{1-j}}{(i, m_0, m_1)^{\mathcal{M}} \longrightarrow_{\texttt{M}} (i+1, m'_0, m'_1)^{\mathcal{M}}}\ \text{Incr}$$

$$\frac{i : \texttt{DECJ}(j,k) \qquad m_j > 0 \qquad m'_j = m_j - 1 \qquad m'_{1-j} = m_{1-j}}{(i, m_0, m_1)^{\mathcal{M}} \longrightarrow_{\texttt{M}} (i+1, m'_0, m'_1)^{\mathcal{M}}}\ \text{Decr}$$

$$\frac{i : \texttt{DECJ}(j,k) \qquad m_j = 0}{(i, m_0, m_1)^{\mathcal{M}} \longrightarrow_{\texttt{M}} (k, m_0, m_1)^{\mathcal{M}}}\ \text{Jump}$$

The instructions can only manipulate the state. In case of a successful instruction i (increment or decrement), the appropriate counter is updated, and the machine moves to the next instruction $i+1$. Decrementing is only possible when the counter value is not zero; otherwise, the value remains unchanged, and the machine jumps to instruction k.

3 Encoding Minsky Machines in ASP

In this section, we show how to encode the state of a Minsky machine $\mathcal{M}$ into ASP. The encoding shows that ASP is Turing complete, and illustrates how NSTs can express stateful types that govern the behaviour of the encoding at the process level. The starting point for our proposed encoding is $(\!|m|\!)$, the interaction protocol for the representation of the natural number m, discussed in the introduction. Given a state (i, m_0, m_1) of a machine $\mathcal{M}$, we define a configuration in ASP denoted $[\![(i, m_0, m_1)]\!]^{\mathcal{M}}$, whose components are depicted in Figure 3. The correctness of our encoding follows from

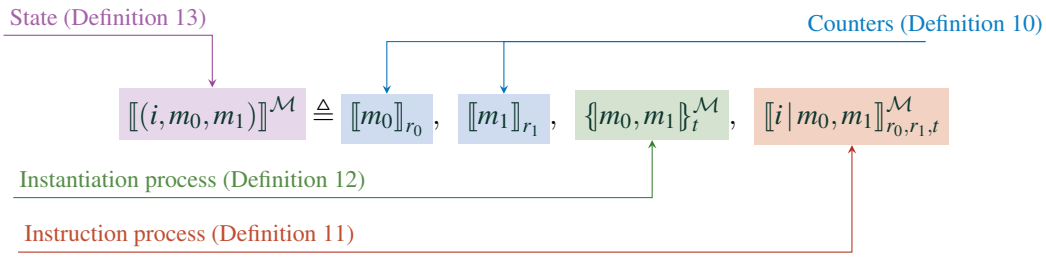


Figure 3: Roadmap to the encoding of Minsky machine state in ASP.

- the *typability* of $[\![(i,m_0,m_1)]\!]^{\mathcal{M}}$ and its components (Theorem 4) and
- *operational correspondence* results, which ensure that (i) every step possible from (i,m_0,m_1) is correctly mimicked by $[\![(i,m_0,m_1)]\!]^{\mathcal{M}}$ (soundness, Theorem 5) and (ii) all steps arising from $[\![(i,m_0,m_1)]\!]^{\mathcal{M}}$ correspond to steps enabled from (i,m_0,m_1) (completeness, Theorem 6).

We start by describing the stateful representation at the level of types, namely the one-counter types that capture the state of the counters (Definition 7), and the *instruction instantiation type*, which enables the continuation of instructions when decrementing (Definition 9). These types provide the appropriate interfaces for typing the process representation of counters $[\![m_i]\!]_{r_i}$ (with $i \in \{0,1\}$), the instantiation process $\{\!|m_0,m_1|\!\}_t^{\mathcal{M}}$, and the encoding of the instruction $[\![i\,|\,m_0,m_1]\!]^{\mathcal{M}}_{r_0,r_1,t}$; these interfaces, in turn, are crucial to enable the safe composition required to obtain the configuration $[\![(i,m_0,m_1)]\!]^{\mathcal{M}}$ given by Definition 13.

Given a Minsky machine $\mathcal{M}$, we will define an associated signature $\Sigma_{\mathcal{M}}$. In giving the different ingredients of the encoding, we will be presenting type and process definitions that we assume are present in $\Sigma_{\mathcal{M}}$. In our descriptions, we shall often write 'machine' rather than 'Minsky machine'.

3.1 Stateful protocols

As anticipated in the introduction, a key idea in our encoding is taking Peano's representation of the natural numbers as a protocol. Such protocols rely on nesting of type parameters in NSTs. We will have two protocols, one for each counter, and so the challenge lies in coordinating their behaviour according to the instructions of the machine.

Definition 7 (Stateful counter protocol)**.** Given a natural number m, its associated counter type is:

$$(\!|m|\!) = \begin{cases} 0[] & \text{if } m = 0 \\ S[(\!|m-1|\!)] & \text{if } m > 0 \end{cases} \tag{1}$$

with type definitions in $\Sigma_{\mathcal{M}}$:

$$0[] = \&\{\texttt{incr}: S[0[]], \texttt{jump}: 0[]\} \qquad \text{(zero)}$$
$$S[\alpha] = \&\{\texttt{incr}: S[S[\alpha]], \texttt{decr}: \alpha\} \qquad \text{(successor)}$$

This protocol is self-explanatory: the choices `incr`, `jump`, and `decr` correspond to the three possible reduction steps for a machine (increments, decrements, jumps), while accounting for successive steps. This is preferable to having two labels corresponding to instructions (with sub-cases for decrement-and-jump). The nesting in types is evident, as is the fact that state does not change in the case of a jump. As a convention, we use type variables α and β as parameters for the first and second counter, respectively.

In principle, given an instruction i of $\mathcal{M}$, we would like to have an encoding of instructions $[\![i]\!]^{\mathcal{M}}_{r_0,r_1}$ with judgement

$$\cdot; r_0 : (\!|m_0|\!), r_1 : (\!|m_1|\!), \cdots \vdash [\![i]\!]^{\mathcal{M}}_{r_0,r_1} :: (x : \mathbf{1})$$

where r_0, r_1 are names where the protocol counters are implemented. Clearly, we do not know the exact type of instruction i, whether it operates on r_0 or r_1, or whether it may result in a jump. Our encoding of instructions must then be *state-aware*. As such we define a process $[\![i \,|\, m_0, m_1]\!]^{\mathcal{M}}_{r_0,r_1}$, which depends on either $0[]$ or $S[\alpha]$ on r_0 and r_1. The issue that arises is that the protocol becomes α in case of a decrement, which is insufficient to capture the state of the counter. Indeed, without extra information, the next instruction cannot be instantiated for the decrementing case, as it depends on $0[]$ or $S[\cdot]$.

To overcome this difficulty, we define an **instantiation protocol** for instructions, able to match α after decrementing a counter as well as to trigger the next instruction, enabling interaction with both counters. As we will see, a corresponding **instantiation process** will act as a coordinator between the process representation of counters and instructions.

To define the instantiation protocol, we first characterise all the instructions that may occur in the possible states of $\mathcal{M}$; this is needed to define $\Sigma_{\mathcal{M}}$ appropriately. We use the following auxiliary notion.

Definition 8. Given a machine $\mathcal{M}$, the set of *accessible instructions* $L_{\mathcal{M}}$ is defined as

$$L_{\mathcal{M}} = \{i+1 \mid (i : I_i) \in \mathcal{M}\} \cup \{k \mid (i : \mathtt{DECJ}(j,k)) \in \mathcal{M}\}$$

The set $L_{\mathcal{M}}$ contains indices for the instructions of $\mathcal{M}$ and possible indices that lie out of bounds of $\mathcal{M}$. It is not the set of all the reachable instructions, but an over-approximation of it. It is easy to verify that the set $L_{\mathcal{M}}$ is finite, as there are only finite instructions in $\mathcal{M}$.

Definition 9 (Instantiation protocol)**.** Given $m_0, m_1 \in \mathbb{N}$ and a machine $\mathcal{M}$, the *instantiation protocol* $\mathcal{T}^{\mathcal{M}}(\!|m_0, m_1|\!)$ is given as:

$$\mathcal{T}^{\mathcal{M}}(\!|m_0, m_1|\!) = \begin{cases} T^{0,0}[] & \text{if } m_0 = m_1 = 0 \\ T^{s,0}[(\!|m_0 - 1|\!)] & \text{if } m_0 > 0 \wedge m_1 = 0 \\ T^{0,s}[(\!|m_1 - 1|\!)] & \text{if } m_0 = 0 \wedge m_1 > 0 \\ T^{s,s}[(\!|m_0 - 1|\!), (\!|m_1 - 1|\!)] & \text{if } m_0 > 0 \wedge m_1 > 0 \end{cases} \tag{2}$$

with the following type definitions in $\Sigma_{\mathcal{M}}$:

$$\begin{aligned}
T^{0,0}[] &= \&\{\, \mathtt{exp}_0 : T^{s,0}[0[]], && \mathtt{exp}_1 : T^{0,s}[0[]], \\
&\qquad \mathtt{dup} : T^{0,0}[] \otimes T^{0,0}[], && \mathtt{ter} : \mathbf{1} \,\} \\
T^{s,0}[\alpha] &= \&\{\, \mathtt{ins}_0 : I[\alpha, 0[]], \\
&\qquad \mathtt{exp}_0 : T^{s,0}[S[\alpha]], && \mathtt{exp}_1 : T^{s,s}[\alpha, 0[]], \\
&\qquad \mathtt{dup} : T^{s,0}[\alpha] \otimes T^{s,0}[\alpha], && \mathtt{ter} : \mathbf{1} \,\} \\
T^{0,s}[\beta] &= \&\{\, \mathtt{ins}_1 : I[0[], \beta], \\
&\qquad \mathtt{exp}_0 : T^{s,s}[0[], \beta], \beta], && \mathtt{exp}_1 : T^{0,s}[S[\beta]], \\
&\qquad \mathtt{dup} : T^{0,s}[\beta] \otimes T^{0,s}[\beta], && \mathtt{ter} : \mathbf{1} \,\} \\
T^{s,s}[\alpha, \beta] &= \&\{\, \mathtt{ins}_0 : I[\alpha, S[\beta]], && \mathtt{ins}_1 : I[S[\alpha], \beta], \\
&\qquad \mathtt{exp}_0 : T^{s,s}[S[\alpha], \beta], && \mathtt{exp}_1 : T^{s,s}[\alpha, S[\beta]], \\
&\qquad \mathtt{ter} : \mathbf{1} && \,\} \\
I[\alpha, \beta] &= \alpha \multimap \beta \multimap \mathbf{1} \multimap \&\{\mathtt{i} : \mathbf{1}\}_{i \in L_{\mathcal{M}}}
\end{aligned}$$

In this definition, the superscript in $T^{-,-}$ denotes the reflection of the zero/successor state of the one-counter types. This agnostic protocol enables the handoff of the one-counter channels to some process abstracting the transition to jump at the process level. The choices that are associated with incrementing and decrementing are called *expansion* (`exp`) and *instantiation* (`ins`), respectively. Expansion predicts the next state, while instantiation simply receives the dependencies of the current instruction and the next instruction. The choices for duplication (`dup`) and termination (`ter`) are for bookkeeping purposes related to the decrementing tree data structure process that the protocol is implemented in.

3.2 Processes

Having presented the nested session types that govern our encoding, we turn our attention to processes. We first give the encoding of the counter process, which closely follows the one-counter protocol; it is a dependency of the instruction process.

Definition 10 (Encoding a counter)**.** The encoding of a counter containing $m \in \mathbb{N}$ into a configuration $\mathcal{S}$ in ASP is given as follows, using Definition 7:

$$[\![m]\!]_r = \begin{cases} \mathtt{proc}(r, R^0) & \text{if } m = 0 \\ [\![m-1]\!]_p, \mathtt{proc}(r, R^s[(\!|m-1|\!)/\alpha]) & \text{if } m > 0 \end{cases} \tag{3}$$

with process definitions in $\Sigma_{\mathcal{M}}$:

$$\cdot \vdash r^0[] = R^0 :: (r : 0[])$$
$$p : \alpha \vdash r^s[\alpha] = R^s :: (r : S[\alpha])$$

and the constituent process definitions:

$$\begin{aligned} R^0 = \mathtt{case}\ r\ (&\mathtt{incr} \Rightarrow c \leftarrow r^0[]; r \leftarrow r^s[0[]]\ c, \\ &\mathtt{jump} \Rightarrow r \leftarrow r^0[]\ ) \\ R^s = \mathtt{case}\ r\ (&\mathtt{incr} \Rightarrow c \leftarrow r^s[\alpha]\ p; r \leftarrow r^s[S[\alpha]]\ c, \\ &\mathtt{decr} \Rightarrow p \leftrightarrow r\ ) \end{aligned}$$

As mentioned before, while $[\![m_0]\!]_{r_0}$ uses type variable α, the definition of $[\![m_1]\!]_{r_1}$ uses β. Each counter is implemented as a stack, following the protocol definition. The continuation process for each choice reflects the construction of the continuation type. For increments, this is the re-instantiation of the current process followed by the successor. For jumps, we only re-instantiate the zero counter process. For decrements, the process providing α is forwarded.

We now define the encoding of instructions: it is state-aware and meant to interact with the instantiation process.

Definition 11 (Instruction processes)**.** Given some instruction $i \in \mathbb{Z}^+$ of $\mathcal{M}$, the encoding of i into a configuration $\mathcal{S}$ in ASP depends on the values of the counters $m_0, m_1 \in \mathbb{N}$; it is given as follows, using Definitions 7 and 9:

$$[\![i \,|\, m_0, m_1]\!]^{\mathcal{M}}_{r_0,r_1,t} = \begin{cases} \mathtt{proc}(x, P_i^{0,0}) & \text{if } m_0 = m_1 = 0 \\ \mathtt{proc}(x, P_i^{s,0}[(\!|m_0-1|\!)/\alpha]) & \text{if } m_0 > 0 \wedge m_1 = 0 \\ \mathtt{proc}(x, P_i^{0,s}[(\!|m_1-1|\!)/\beta]) & \text{if } m_0 = 0 \wedge m_1 > 0 \\ \mathtt{proc}(x, P_i^{s,s}[(\!|m_0-1|\!)/\alpha, (\!|m_1-1|\!)/\beta]) & \text{if } m_0 > 0 \wedge m_1 > 0 \end{cases}$$

$$P_i^{0,0} = \begin{cases} r_0.\texttt{incr};\ t.\texttt{exp}_0;\ x \leftarrow p_{i+1}^{s,0}[0[]]\ r_0\ r_1\ t\ y & \text{if } i : \texttt{INC}(0) \\ r_1.\texttt{incr};\ t.\texttt{exp}_1;\ x \leftarrow p_{i+1}^{0,s}[0[]]\ r_0\ r_1\ t\ y & \text{if } i : \texttt{INC}(1) \\ r_j.\texttt{jump};\ x \leftarrow p_k^{0,0}[]\ r_0\ r_1\ t\ y & \text{if } i : \texttt{DECJ}(j,k) \\ \texttt{wait}\ y; Q_i^{0,0} & \text{if } i > n \end{cases}$$

$$P_i^{s,0} = \begin{cases} r_0.\texttt{incr};\ t.\texttt{exp}_0;\ x \leftarrow p_{i+1}^{s,0}[S[\alpha]]\ r_0\ r_1\ t\ y & \text{if } i : \texttt{INC}(0) \\ r_1.\texttt{incr};\ t.\texttt{exp}_1;\ x \leftarrow p_{i+1}^{s,s}[\alpha,0[]]\ r_0\ r_1\ t\ y & \text{if } i : \texttt{INC}(1) \\ r_0.\texttt{decr};\ t.\texttt{ins}_0;\ \texttt{send}\ t\ r_0; \texttt{send}\ t\ r_1; \texttt{send}\ t\ y; t.(\texttt{i}+1);\ t \leftrightarrow x & \text{if } i : \texttt{DECJ}(0,k) \\ r_1.\texttt{jump};\ x \leftarrow p_k^{s,0}[\alpha]\ r_0\ r_1\ t\ y & \text{if } i : \texttt{DECJ}(1,k) \\ \texttt{wait}\ y; Q_i^{s,0} & \text{if } i > n \end{cases}$$

$$P_i^{0,s} = \begin{cases} r_0.\texttt{incr};\ t.\texttt{exp}_0;\ x \leftarrow p_{i+1}^{s,s}[0[],\beta]\ r_0\ r_1\ t\ y & \text{if } i : \texttt{INC}(0) \\ r_1.\texttt{incr};\ t.\texttt{exp}_1;\ x \leftarrow p_{i+1}^{0,s}[S[\beta]]\ r_0\ r_1\ t\ y & \text{if } i : \texttt{INC}(1) \\ r_0.\texttt{jump};\ x \leftarrow p_k^{0,s}[\beta]\ r_0\ r_1\ t\ y & \text{if } i : \texttt{DECJ}(0,k) \\ r_1.\texttt{decr};\ t.\texttt{ins}_1;\ \texttt{send}\ t\ r_0; \texttt{send}\ t\ r_1; \texttt{send}\ t\ y; t.(\texttt{i}+1);\ t \leftrightarrow x & \text{if } i : \texttt{DECJ}(1,k) \\ \texttt{wait}\ y; Q_i^{0,s} & \text{if } i > n \end{cases}$$

$$P_i^{s,s} = \begin{cases} r_0.\texttt{incr};\ t.\texttt{exp}_0;\ x \leftarrow p_{i+1}^{s,s}[S[\alpha],\beta]\ r_0\ r_1\ t\ y & \text{if } i : \texttt{INC}(0) \\ r_1.\texttt{incr};\ t.\texttt{exp}_1;\ x \leftarrow p_{i+1}^{s,s}[\alpha,S[\beta]]\ r_0\ r_1\ t\ y & \text{if } i : \texttt{INC}(1) \\ r_j.\texttt{decr};\ t.\texttt{ins}_j;\ \texttt{send}\ t\ r_0; \texttt{send}\ t\ r_1; \texttt{send}\ t\ y; t.(\texttt{i}+1);\ t \leftrightarrow x & \text{if } i : \texttt{DECJ}(j,k) \\ \texttt{wait}\ y; Q_i^{s,s} & \text{if } i > n \end{cases}$$

Figure 4: Constituent processes used in the encoding of instructions (Definition 11).

with the process definitions in $\Sigma_{\mathcal{M}}$ using Definition 8:

$$\begin{aligned} &r_0 : 0[], \quad r_1 : 0[], \quad t : T^{0,0}[], \quad && y : \mathbf{1} \vdash p_{i \in L_{\mathcal{M}}}^{0,0}[] && = P_i^{0,0} :: (x : \mathbf{1}) \\ &r_0 : S[\alpha], r_1 : 0[], \quad t : T^{s,0}[\alpha], \quad && y : \mathbf{1} \vdash p_{i \in L_{\mathcal{M}}}^{s,0}[\alpha] && = P_i^{s,0} :: (x : \mathbf{1}) \\ &r_0 : 0[], \quad r_1 : S[\beta], t : T^{0,s}[\beta], \quad && y : \mathbf{1} \vdash p_{i \in L_{\mathcal{M}}}^{0,s}[\beta] && = P_i^{0,s} :: (x : \mathbf{1}) \\ &r_0 : S[\alpha], r_1 : S[\beta], t : T^{s,s}[\alpha,\beta], \quad && y : \mathbf{1} \vdash p_{i \in L_{\mathcal{M}}}^{s,s}[\alpha,\beta] && = P_i^{s,s} :: (x : \mathbf{1}) \end{aligned}$$

and the constituent process definitions are given in Figure 4.

In this definition, the subscripts r_0, r_1, and t in the encoding refer to the internal renaming of the channels of the counter processes 0 and 1 and the instantiation process, in that order. Channels x and y never reduce; hence, no renaming is necessary.

The definitions for the instruction processes deal with branching by being state-aware, which is a consequence of the stateful protocols; an instruction must know when it can decrement or when it should jump, and deal with the subsequent state accordingly. This highlights the need for the instantiation protocol/process: without it, the instruction cannot move forward.

In Figure 4, the instructions are implemented by sending the appropriate label associated with the current instruction action over the corresponding counter channel. For incrementing and jumping, it is possible to predict the next state; hence, we can instantiate the next instruction directly; for example, in

the encoding of increment instructions as defined in $P_i^{0,0}$. Note that when incrementing, the state of the counters is synchronised with the instantiation process, which is not necessary for jumping, as the state remains unchanged.

The case for decrementing is slightly different after sending the label, as it is impossible to instantiate the next instruction over a generic type variable. Instead, the current instruction hands off the dependent processes and the next instruction index to the instantiation process, which closes the circle and allows the definitions to be well-typed; this occurs, for example, in the encoding of decrement-jump in $P_i^{s,s}$.

Lastly, there is a special case for out-of-bounds instructions. In this case, the machine cannot reduce further and has terminated. To simulate this, we use the external channel y, which halts further progress; for example, in the encoding of out-of-bound i in $P_i^{0,0}$. The continuation process Q_i can be any process as long as it is distinct for different i, to capture the final state for each possible value of i.

Similar to Definition 10, process definitions are added to the global environment $\Sigma_{\mathcal{M}}$ to move to instructions and make use of $L_{\mathcal{M}}$ (Definition 8) to include the finite set of accessible instructions. Unlike the process definitions, which are defined on $L_{\mathcal{M}}$, the encoding of an instruction is unbounded.

To continue to the next instruction after decrementing, we rely on the instantiation process, defined next. The definitions make use of a decrementing tree data structure, as it is the most straightforward both in implementation and in understanding the bookkeeping involved.

Definition 12 (Instantiation process)**.** Given some $m_0, m_1 \in \mathbb{N}$, the *instantiation process* $\{\!|m_0, m_1|\!\}_t^{\mathcal{M}}$ is a configuration $\mathcal{S}$ in ASP defined as follows, using Definitions 7, 9 and 11:

$$\{\!|m_0,m_1|\!\}_z^{\mathcal{M}} = \begin{cases} \mathtt{proc}(z, N^{0,0}) & \text{if } m_0 = m_1 = 0 \\ \{\!|0,0|\!\}_a^{\mathcal{M}},\ \mathtt{proc}(z, N^{1,0}) & \text{if } m_0 = 1 \wedge m_1 = 0 \\ \{\!|m-1,0|\!\}_a^{\mathcal{M}},\ \mathtt{proc}(z, N^{s,0}[(\!|m_0-2|\!)/\alpha]) & \text{if } m_0 > 1 \wedge m_1 = 0 \\ \{\!|0,0|\!\}_b^{\mathcal{M}},\ \mathtt{proc}(z, N^{0,1}) & \text{if } m_0 = 0 \wedge m_1 = 1 \\ \{\!|0,m_1-1|\!\}_b^{\mathcal{M}},\ \mathtt{proc}(z, N^{0,s}[(\!|m_1-2|\!)/\beta]) & \text{if } m_0 = 0 \wedge m_1 > 1 \\ \{\!|0,1|\!\}_a^{\mathcal{M}},\ \{\!|1,0|\!\}_b^{\mathcal{M}},\ \mathtt{proc}(z, N^{1,1}) & \text{if } m_0 = m_1 = 1 \\ \{\!|m_0-1,1|\!\}_a^{\mathcal{M}},\ \{\!|m_0,0|\!\}_b^{\mathcal{M}},\ \mathtt{proc}(z, N^{s,1}[(\!|m_0-2|\!)]/\alpha]) & \text{if } m_0 > 1 \wedge m_1 = 1 \\ \{\!|0,m_1|\!\}_a^{\mathcal{M}},\ \{\!|1,m_1-1|\!\}_b^{\mathcal{M}},\ \mathtt{proc}(z, N^{1,s}[(\!|m_1-2|\!)/\beta]) & \text{if } m_0 = 1 \wedge m_1 > 1 \\ \{\!|m_0-1,m_1|\!\}_a^{\mathcal{M}},\ \{\!|m_0,m_1-1|\!\}_b^{\mathcal{M}}, & \text{if } m_0 > 1 \wedge m_1 > 1 \\ \quad \mathtt{proc}(z, N^{s,s}[(\!|m_0-2|\!)/\alpha, (\!|m_1-2|\!)/\beta]) & \end{cases} \tag{4}$$

with the process definitions in $\Sigma_{\mathcal{M}}$:

$$\begin{array}{llll} \cdot;\cdot & & \vdash t^{0,0}[] & = N^{0,0} :: (z : T^{0,0}[]) \\ \cdot; a : T^{0,0}[] & & \vdash t^{1,0}[] & = N^{1,0} :: (z : T^{s,0}[0[]]) \\ \alpha; a : T^{s,0}[\alpha] & & \vdash t^{s,0}[\alpha] & = N^{s,0} :: (z : T^{s,0}[S[\alpha]]) \\ \cdot; & b : T^{0,0}[] & \vdash t^{0,1}[] & = N^{0,1} :: (z : T^{0,s}[0[]]) \\ \beta; & b : T^{0,s}[\beta] & \vdash t^{0,s}[\beta] & = N^{0,s} :: (z : T^{0,s}[S[\beta]]) \\ \cdot; a : T^{0,s}[0[]], & b : T^{s,0}[0[]] & \vdash t^{1,1} & = N^{1,1} :: (z : T^{s,s}[0[], 0[]]) \\ \alpha; a : T^{s,s}[\alpha, 0[]], & b : T^{s,0}[S[\alpha]] & \vdash t^{s,1}[\alpha] & = N^{s,1} :: (z : T^{s,s}[S[\alpha], 0[]]) \\ \beta; a : T^{0,s}[\beta], & b : T^{s,s}[0[], S[\beta]] & \vdash t^{1,s}[\beta] & = N^{1,s} :: (z : T^{s,s}[0[], S[\beta]]) \\ \alpha, \beta; a : T^{s,s}[\alpha, S[\beta]], & b : T^{s,s}[S[\alpha], \beta] & \vdash t^{s,s}[\alpha, \beta] & = N^{s,s} :: (z : T^{s,s}[S[\alpha], S[\beta]]) \end{array}$$

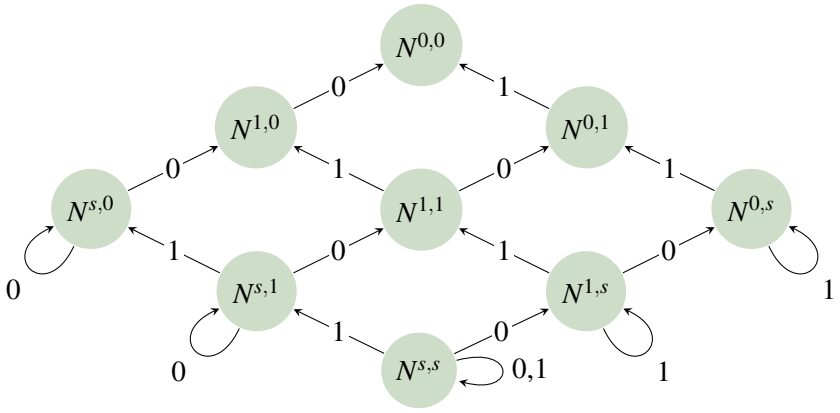


Figure 5: Instantiation process (Definition 12): Dependencies between constituent process definitions indicating decrements over either the first (0) or the second (1) counter.

Figure 6 gives selected cases of the constituent processes definitions and Figure 5 presents a dependency graph. A full account is given in Appendix B.1.

The instantiation process $\{\!|m_0, m_1|\!\}_i^{\mathcal{M}}$ implements a tree-like structure where processes $N^{-,-}$ represent *nodes*. The encoding builds naturally on the one-counter protocols, creating dependencies on states of decrements on the counters forming the tree structure, where the channels a and b denote whether the first or second counter is decremented, respectively. The dependencies of the composition are presented in Figure 5. Here, the superscript notation of Definition 9 is reused with the addition of 1. In the context of the protocol, this annotation was used merely to denote state; in this definition, it is expanded to reflect the transitions between states.

- 0 refers to a zero state without transition;
- 1 refers to the transition from a successor to a zero state;
- s refers to the transition from a successor to a successor state.

Transitions are thus implemented as a dependency on a type definition, unlike the counter, which uses a generic type parameter for its dependency. Consequently, these processes are instantiated using the second-order predecessor, where counters and instructions depend on the first-order predecessor.

In Figure 6 we present the most representative constituent process nodes of the instantiation process, with the remaining nodes following similar constructions. We can differentiate nodes by the number of decrementing dependencies in the tree: leaf nodes where both counters are zero, linear nodes with one non-zero counter and binary nodes with both counters non-zero. We comment on some of them:

- Node $N^{0,0}$ is a leaf that just captures the zero state for both counters. This node exists primarily due to the infinite recursive nature of the one-counters and forms the starting point from which the decrementing tree is constructed, both in the encoding and in the simulation.
- Node $N^{1,0}$ is a linear node that captures the state of 1 for the first counter and the transition to 0. All states are constant, and therefore, it uses only closed types. The non-zero state implements instantiation to enable progress after decrement, which simply carries over the dependencies of the previous instruction, followed by the instantiation of the next instruction as designated by the current instruction. In case of an increment on the zero state, the node becomes binary by creating a new branch, creating a duplicate to create the choices. Because the node represents a zero state

$$
\begin{aligned}
N^{0,0} = \mathtt{case}\ z\ (&\mathtt{exp}_0 \Rightarrow c \leftarrow t^{0,0}[];z \leftarrow t^{1,0}[]\ c\\
&\mathtt{exp}_1 \Rightarrow c \leftarrow t^{0,0}[];z \leftarrow t^{0,1}[]\ c\\
&\mathtt{dup} \Rightarrow c \leftarrow t^{0,0}[];\mathtt{send}\ z\ c;z \leftarrow t^{0,0}[]\\
&\mathtt{ter} \Rightarrow \mathtt{close}\ z)\\
N^{1,0} = \mathtt{case}\ z\ (&\mathtt{ins}_0 \Rightarrow r_0 \leftarrow \mathtt{recv}\ z;r_1 \leftarrow \mathtt{recv}\ z;y \leftarrow \mathtt{recv}\ z;\\
&\qquad \mathtt{case}\ z\ (\mathtt{i} \Rightarrow z \leftarrow p_i^{0,0}[]\ r_0\ r_1\ a\ y)_{i\in L\mathcal{M}}\\
&\mathtt{exp}_0 \Rightarrow a.\mathtt{exp}_0;\ z \leftarrow t^{s,0}[0[]]\ a\\
&\mathtt{exp}_1 \Rightarrow a.\mathtt{dup};\ b \leftarrow \mathtt{recv}\ a;a.\mathtt{exp}_1;\ b.\mathtt{exp}_0;\ z \leftarrow t^{1,1}[]\ a\ b\\
&\mathtt{dup} \Rightarrow a.\mathtt{dup};\ a' \leftarrow \mathtt{recv}\ a;c \leftarrow t^{1,0}[]\ a;\mathtt{send}\ z\ c;z \leftarrow t^{1,0}[]\ a'\\
&\mathtt{ter} \Rightarrow a.\mathtt{1};\ \mathtt{wait}\ a;\mathtt{close}\ z)\\
N^{s,s} = \mathtt{case}\ z\ (&\mathtt{ins}_0 \Rightarrow b.\mathtt{ter};\ \mathtt{wait}\ b;r_0 \leftarrow \mathtt{recv}\ z;r_1 \leftarrow \mathtt{recv}\ z;y \leftarrow \mathtt{recv}\ z;\\
&\qquad \mathtt{case}\ z\ (\mathtt{i} \Rightarrow z \leftarrow p_i^{s,s}[\alpha,\beta]\ r_0\ r_1\ a\ y)_{i\in L\mathcal{M}}\\
&\mathtt{ins}_1 \Rightarrow a.\mathtt{ter};\ \mathtt{wait}\ a;r_0 \leftarrow \mathtt{recv}\ z;r_1 \leftarrow \mathtt{recv}\ z;y \leftarrow \mathtt{recv}\ z;\\
&\qquad \mathtt{case}\ z\ (\mathtt{i} \Rightarrow z \leftarrow p_i^{s,s}[\alpha,\beta]\ r_0\ r_1\ b\ y)_{i\in L\mathcal{M}}\\
&\mathtt{exp}_0 \Rightarrow a.\mathtt{exp}_0;\ b.\mathtt{exp}_0;\ z \leftarrow t^{s,s}[S[\alpha],\beta]\ a\ b\\
&\mathtt{exp}_1 \Rightarrow a.\mathtt{exp}_1;\ b.\mathtt{exp}_1;\ z \leftarrow t^{s,s}[\alpha,S[\beta]]\ a\ b\\
&\mathtt{ter} \Rightarrow a.\mathtt{ter};\ b.\mathtt{ter};\ \mathtt{wait}\ a;\mathtt{wait}\ b;\mathtt{close}\ z)
\end{aligned}
$$

Figure 6: Examples of leaf, linear and binary nodes of the instantiation process (Definition 12).

for a counter and consequently is linear, a possible parent can represent a zero state for the same counter and be linear. Hence, it is necessary to implement duplication so the parent can introduce new branches.

- Node $N^{s,s}$ is a binary node that captures the state of incrementing state for both counters and the transition to a successor state for both. All states are variable expressions, and therefore, it uses open types; the process still needs to be instantiated with concrete values. For instantiation, it makes use of termination to remove the unused branch, as residual processes need to be terminated in ASP. Furthermore, because this is already a binary node, no new branches have to be created by either themselves or their parents; hence no implementation of duplication.

To manage the tree, process definitions are added to the global environment $\Sigma_{\mathcal{M}}$. Note that Definition 12 depends on the process definitions given in Definition 11.

Figure 7 illustrates how the decrement tree branches. The root node starts by duplicating its branch and then updating each branch using expansion to represent the different ways of decrementing. In turn, the branch with non-zero states will do the same, while the branch with a zero state will remain linear and forward the expansion. The expansions are terminated in the leaf node that becomes linear and creates a new leaf node.

We are finally ready to encode the state of a machine, building upon the previous definitions.

Definition 13 (Encoding state)**.** Using Definitions 10 to 12, the encoding of state for machine $\mathcal{M}$ is

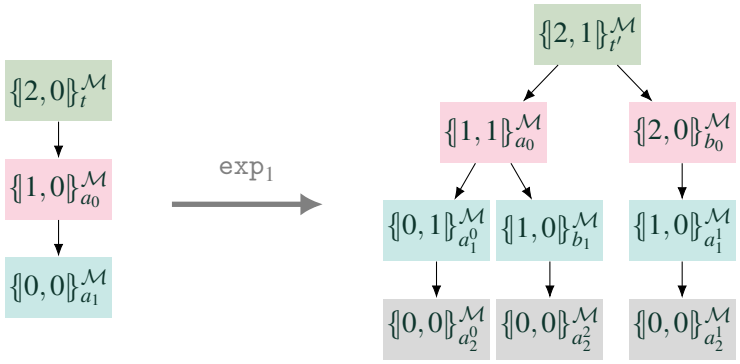


Figure 7: Example of branching of a linear node in Definition 12.

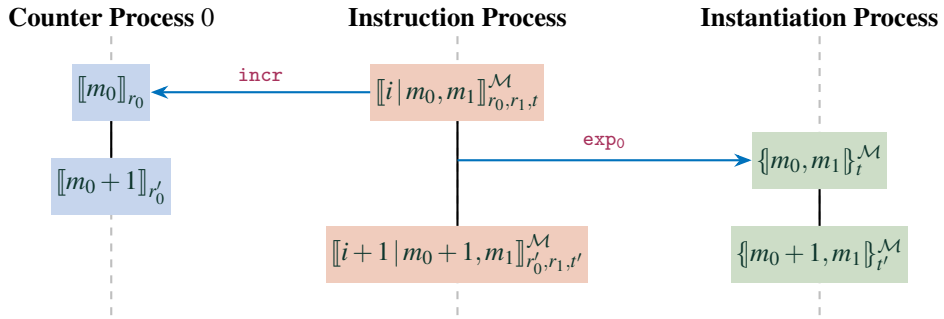


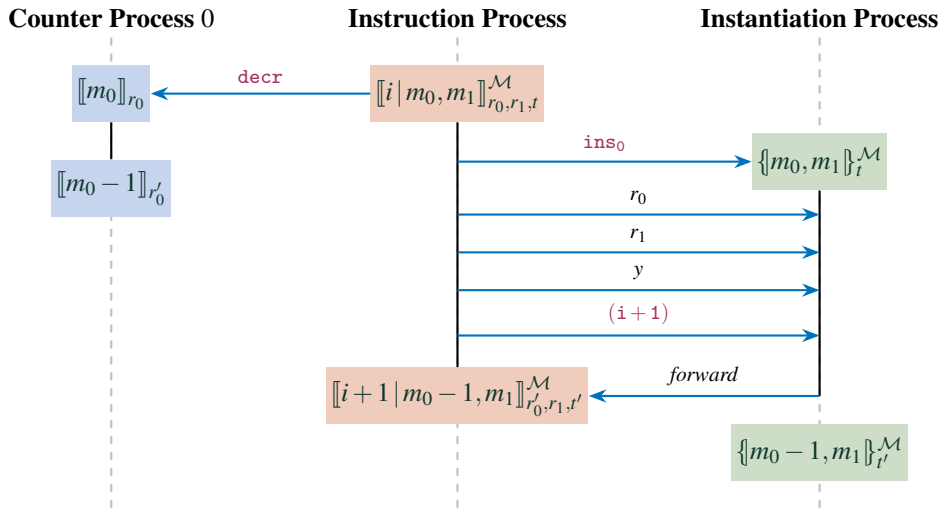
Figure 8: An increment on counter 0.

defined as:

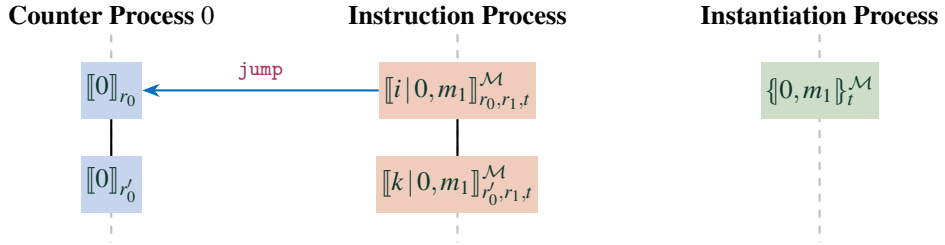
$$[\![(i, m_0, m_1)]\!]^{\mathcal{M}} = [\![m_0]\!]_{r_0}, [\![m_1]\!]_{r_1}, \{\!|m_0, m_1|\!\}_t^{\mathcal{M}}, [\![i \mid m_0, m_1]\!]_{r_0,r_1,t}^{\mathcal{M}} \tag{5}$$

We use sequence diagrams to explain the encoding of the three possible steps of a machine.

1. Figure 8 shows how an incrementing reduction is mimicked. In the encoding, each process is able to predict the next state. Consequently, the instruction process simply sends the instruction to the counter (as denoted by the arrow with label `incr`) and then synchronises that change to the instantiation process (as denoted by the arrow with label $\exp_0$). Then, each process can move such that it matches the next state of the encoding. However, due to the concurrent, asynchronous nature of ASP, some processes (in particular the instruction process) can move ahead. Due to the properties of ASP, like type preservation, lagging processes are able to catch up without side effects, which only means that the encoding simply skipped a step but is still consistent.
2. Figure 9a shows the representation of a decrement. Unlike incrementing, the instruction process is not able to predict the next concrete state; it relies on the instantiation process to move to the next instruction. This is highlighted by the sending of the dependencies, followed by the forward, where the root node takes the place of the current instruction. Furthermore, a branch is promoted to become the new root node. Unlike incrementing, the instruction process has to wait for the instantiation process to catch up before it can proceed to instantiate the next instruction.
3. Finally, Figure 9b shows how we mimic jumping. Similar to the incrementing instruction, each



(a) A decrement: Counter 0 has value $m_0 > 0$.



(b) A jump: Counter 0 has value $m_0 = 0$.

Figure 9: Encoding of a decrement-and-jump $i : \mathtt{DECJ}(0,k)$ on counter 0, as sequence diagrams.

process can predict the next state; in this case, however, the state does not change. Instead, the instruction is simply sent, and afterwards we move to the next instruction k.

3.3 Correctness

Before illustrating our encoding (which we do in Section 3.4), we address its correctness. The first step is to prove that the encoding is well-typed in ASP; hence, it inherits type preservation, progress, and confluence.

Theorem 4 (Typability)**.** *Given any state* $(i,m_0,m_1)^{\mathcal{M}}$ *we have:*

- $\cdot \models_{\Sigma_{\mathcal{M}}} [\![m_i]\!]_r :: (r : (\!|m_i|\!))$ *(for* $i \in \{0,1\}$*; cf. Definitions 7 and 10);*
- $\cdot \models_{\Sigma_{\mathcal{M}}} \{\!|m_0,m_1|\!\}^{\mathcal{M}}_t :: (t : \mathcal{T}^{\mathcal{M}}(\!|m_0,m_1|\!))$ *(cf. Definitions 9 and 12);*
- $r_0 : (\!|m_0|\!), r_1 : (\!|m_1|\!), t : \mathcal{T}^{\mathcal{M}}(\!|m_0,m_1|\!), y : \mathbf{1} \models_{\Sigma_{\mathcal{M}}} [\![i \mid m_0,m_1]\!]^{\mathcal{M}}_{r_0,r_1,t} :: (x : \mathbf{1})$ *(cf Definition 11).*

Hence, by Definition 13:

$$y : \mathbf{1} \models_{\Sigma_{\mathcal{M}}} [\![(i,m_0,m_1)]\!]^{\mathcal{M}} :: (x : \mathbf{1}).$$

Well-typedness of the encoding ensures not only that the properties of ASP hold, but also that the encoding follows the protocols for the individual components, as correctness is partially captured on the

level of types. Theorem 4 is proved by induction on the value of the counter and case analysis on the instruction; see [22] for the full typing derivations.

After establishing typability, the second step is to prove correspondence on the level of reductions. To prove such operational correspondence, we show that the semantics of the two-counter machine are simulated correctly (soundness) and that the encoding does not introduce any side effects (completeness).

Theorem 5 (Soundness)**.** *Let (i,m_0,m_1) be a state for a machine $\mathcal{M}$. If $(i,m_0,m_1) \longrightarrow_{\mathtt{M}} (i',m_0',m_1')$, then $[\![(i,m_0,m_1)]\!]^{\mathcal{M}} \longrightarrow^+ [\![(i',m_0',m_1')]\!]^{\mathcal{M}}$.*

The proof of Theorem 5 follows largely from the construction of the stateful session types, which capture the semantics with regard to transitions of state for different instructions. As the encoding is well-typed (Theorem 4) as long as the correct instruction is sent in correspondence with the current state i, the encoding behaves the same as the machine concerning soundness.

This leaves the validation of the correct instruction being executed. This is relatively easy to verify, as in the encoding, each instruction is prefixed with the sending of a correct label over the correct channel, with no further interaction between the instruction and counter process until the next instruction. The correctness of the next instruction follows in cases of incrementing and jumping, as the next instruction is immediately instantiated. The case of decrementing is different, as it makes use of the instantiation process; however, as such an instruction sends the label for the next instruction and the instantiation process takes this label to instantiate the correct next instruction, it is clear that, from this perspective, the encoding behaves correctly.

Because the encoding is well-typed, progress is guaranteed except over channel y, which is there to simulate termination for out-of-bound instructions.

The proof can be further reinforced by the sequence diagrams Figures 9a to 9b. Here, it is possible to directly choose the reductions that lead to the next state of an encoding such that it matches the reduction of the two-counter machine in one step.

Theorem 6 (Completeness)**.** *Suppose $[\![(i,m_0,m_1)]\!]^{\mathcal{M}} \longrightarrow^+ \mathcal{S}$ where $\mathcal{S}$ is a configuration in ASP. Then there exists (i',m_0',m_1') such that $\mathcal{S} \longrightarrow^+ [\![(i',m_0',m_1')]\!]^{\mathcal{M}}$ and $(i,m_0,m_1) \longrightarrow_{\mathtt{M}}^+ (i',m_0',m_1')$.*

The proof of Theorem 6 follows the same structure as the proof of soundness (Theorem 5), using the fact that ASP is confluent. Essentially, if a translation of the machine state reduces to some configuration $\mathcal{S}$, then $\mathcal{S}$ can further reduce to a translation of another machine state. This follows from reordering of the reductions as a consequence of confluence and soundness.

As mentioned in the discussion of the sequence diagrams, it is entirely possible for a process like that of the instruction to move ahead without waiting for the counter to match the encoding first. In the context of ASP each of these reductions can always be applied at a later state to match the encoding; at no point can other processes influence the reduction to the next state for such a process. Hence, again, this is simply a reordering of the order of reductions that was found in the proof of soundness.

3.4 Examples

To illustrate the Soundness and Completeness theorems, and to discuss the behaviour of the encoding, we consider the translation of a simple machine $\mathcal{M}$:

$$\{(1 : \mathtt{INC}(1)), (2 : \mathtt{INC}(0)), (3 : \mathtt{DECJ}(1,2))\}.$$

Depending on the starting instruction, $\mathcal{M}$ can exhibit both terminating and non-terminating behaviour:

- Terminating: $(1,0,0) \longrightarrow_{\mathtt{M}} (2,0,1) \longrightarrow_{\mathtt{M}} (3,1,1) \longrightarrow_{\mathtt{M}} (4,1,0) \not\longrightarrow_{\mathtt{M}}$

- Non-terminating: $(2,0,0) \longrightarrow_{\mathtt{M}} (3,1,0) \longrightarrow_{\mathtt{M}} (2,1,0) \longrightarrow_{\mathtt{M}} \cdots$

The first example terminates because of the initial increment, which allows the decrement instruction to succeed and move out-of-bounds. In the second example, this initial increment is skipped, and as a consequence, the decrement fails and jumps back to the initial instruction, allowing it to loop continuously.

Using the sequence diagrams for each step in the machine (Figures 8 and 9), a set of reductions in the encoding can be composed such that it leads to the encoded next state, which coincides with the property of soundness. The difference between the two examples is that the terminating state makes use of the instantiation process to move to the next instruction. The non-terminating state makes no use of the instantiation process as the decrementing fails, and the state does not change.

Because of the asynchronous nature of ASP, the instruction process in the encoding can move independently from the other processes until it requires something of them in case of decrements. This can be illustrated by looking at the terminating example:

$$\begin{aligned}[\![(1,0,0)]\!]^{\mathcal{M}} \longrightarrow^{+} & [\![0]\!]_{r_0}, \mathtt{msg}(r_0', r_0.\mathtt{incr};\ r_0' \leftrightarrow r_0),\\ & [\![0]\!]_{r_1}, \mathtt{msg}(r_1', r_1.\mathtt{incr};\ r_1' \leftrightarrow r_1),\\ & \{\!|0,0|\!\}^{\mathcal{M}}_{t}, \mathtt{msg}(t', t.\mathtt{exp}_1;\ t' \leftrightarrow t), \mathtt{msg}(t^2, t^1.\mathtt{exp}_0;\ t^2 \leftrightarrow t^1), [\![3 \,|\, 1,1]\!]^{\mathcal{M}}_{r_0',r_1',t^2}\end{aligned}$$

where $\longrightarrow^{+}$ denotes to the transitive closure of $\longrightarrow$. Here, the encoding of the instruction is already at the state $(3,1,1)$. At this point, the consumption of all messages would allow the encoding to match for all other processes. However, there is nothing preventing the process from not doing so, except when the instruction reduces further.

$$[\![3 \,|\, 1,1]\!]^{\mathcal{M}}_{r_0',r_1',t^2} \longrightarrow^{+} \mathtt{msg}(r_0'', r_0'.\mathtt{decr};\ r_0'' \leftrightarrow r_0'), \mathtt{msg}(t^3, t^2.\mathtt{ins}_0;\ t^3 \leftrightarrow t^2), \cdots, \mathtt{msg}(x, t.4;\ x \leftrightarrow t)$$

Because the protocol allows for a successful decrement, the current instruction hands over to the instruction process, as that process is aware of the state after decrementing the counter; this is done by sending the dependencies, the label for the next instruction and the channel that the instruction process offers. The only reductions possible are those of the consumption of messages, in particular those of the instantiation process, as they construct the next instruction.

$$\begin{aligned}&\longrightarrow^{+} \{\!|1,1|\!\}^{\mathcal{M}}_{t^2}, \mathtt{msg}(t^3, t^2.\mathtt{ins}_0;\ t^3 \leftrightarrow t^2), \cdots, \mathtt{msg}(x, t.4;\ x \leftrightarrow t)\\ &\quad = \{\!|0,1|\!\}^{\mathcal{M}}_{t_0}, \{\!|1,0|\!\}^{\mathcal{M}}_{t_1}, \mathtt{proc}(t_2, N^{1,1}), \mathtt{msg}(t^3, t^2.\mathtt{ins}_0;\ t^3 \leftrightarrow t^2), \cdots, \mathtt{msg}(x, t.4;\ x \leftrightarrow t)\\ &\longrightarrow^{+} \{\!|0,1|\!\}^{\mathcal{M}}_{t_0}, [\![4 \,|\, 0,1]\!]^{\mathcal{M}}_{r_0'',r_1',t_0}\end{aligned}$$

When we skip ahead of the instantiation (for branching see Figure 7), then what is left is that, for the process to terminate, the branch $1,0$ and instantiate the next instruction using the branch $0,1$ and the dependencies of the previous instruction.

In this case, instruction 4 is out-of-bounds: the instruction process becomes poised over channel y; when the reductions of the counters take place, the encoding matches the state $(4,0,1)$.

In the case of non-termination, this final synchronisation never happens; moreover, it never decrements, and so the instruction never makes use of the instantiation process. This means that increment and jump messages to the counter and the expand messages to the instantiation process just queue up. But at any point, they can be consumed such that the encoding can match the state of the simulated machine.

Notable in this example is that instructions never require anything from the counter and merely update their state. While at the process level the counters serve little function, their use comes from the session types they implement, enforcing the correct behaviour of processes using them. They essentially represent the possible reductions with state transitions independent of the definition of the instructions.

4 Concluding Remarks

In this paper, we have considered the expressiveness of ASP, the asynchronous process language associated with the Nested Session Types by Das *et al.*. Starting from an intuitive encoding of numbers that naturally leverages nested types that implement Peano's representation of the naturals (Definition 7), we encoded Minsky machines by developing instruction processes that are state-aware in order to precisely and correctly enact decrement-and-jump instructions. The correctness properties of our encoding are given in terms of typability (Theorem 4) and operational correspondence (soundness and completeness guarantees, cf. Theorems 5 and 6). Combined, these results entail that the coupling of ASP and NSTs is Turing complete. In future work, we would like to explore connections between ASP (and NSTs) with other typed languages with logical foundations, such as CLASS [19, 7], which features both polymorphism and state.

Related Work Closely related work has been discussed in the introduction; here we comment on other relevant literature. Minsky machines are a convenient model for proving the Turing completeness of models of concurrency. They were used, for instance, by Busi *et al.* [4, 5, 6] to compare different forms of infinite behaviour in CCS and by Lanese *et al.* [16, 17] to establish the expressiveness of higher-order process calculi. In both cases, the calculi are untyped. In typed process calculi, the situation is different, as types enforce correctness by considerably narrowing down the kind of (non-terminating) behaviours that well-typed processes can express. We are not aware of other typed encodings of Minsky machines.

In linearly-typed calculi, the limited flexibility is even more prominent, as linearity induces properties such as confluence and deadlock freedom. This is the case for typed frameworks derived from the Curry-Howard correspondence between session types and linear logic [8], which are strongly normalising. In this respect, the work of Balzer *et al.* [3] is related: they consider a session-typed π-calculus with *manifest sharing*, and show that adding recursive shared types to an otherwise linearly-typed language suffices to correctly encode the untyped, asynchronous π-calculus.

Also in the typed setting, Kouzapas *et al.* [14, 15] obtain a series of relative expressiveness results concerning first- and higher-order π-calculi with session types (not based on linear logic). They discover that non-tail recursive types are essential to encode a name-passing calculus into a higher-order calculus. From this perspective, advanced type disciplines such as NSTs bring substantial expressive power, beyond usual recursive types [11]; still, it is interesting to note that correctly encoding Minsky machines is not trivial, even when counting with the power of (nested) recursion and polymorphism in types.

Acknowledgments We are grateful to the anonymous reviewers for their constructive feedback. This research has been partially supported by the Dutch Research Council (NWO) under project "Cyclic Structures in Programs and Proofs" (OCENW.XL.23.089).

References

[1] Bernardo Almeida, Andreia Mordido, Peter Thiemann & Vasco T. Vasconcelos (2022): *Polymorphic lambda calculus with context-free session types*. *Inf. Comput.* 289(Part), p. 104948, doi:10.1016/J.IC.2022.104948. Available at `https://doi.org/10.1016/j.ic.2022.104948`.

[2] Bernardo Almeida, Andreia Mordido & Vasco T. Vasconcelos (2020): *Deciding the Bisimilarity of Context-Free Session Types*. In Armin Biere & David Parker, editors: *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings,*

Part II, *Lecture Notes in Computer Science* 12079, Springer, pp. 39–56, doi:10.1007/978-3-030-45237-7_3. Available at https://doi.org/10.1007/978-3-030-45237-7_3.

[3] Stephanie Balzer, Frank Pfenning & Bernardo Toninho (2018): *A Universal Session Type for Untyped Asynchronous Communication*. In Sven Schewe & Lijun Zhang, editors: *29th International Conference on Concurrency Theory, CONCUR 2018, Beijing, China, September 4-7, 2018*, *LIPIcs* 118, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 30:1–30:18, doi:10.4230/LIPICS.CONCUR.2018.30. Available at https://doi.org/10.4230/LIPIcs.CONCUR.2018.30.

[4] Nadia Busi, Maurizio Gabbrielli & Gianluigi Zavattaro (2003): *Replication vs. Recursive Definitions in Channel Based Calculi*. In Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow & Gerhard J. Woeginger, editors: *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proceedings*, *Lecture Notes in Computer Science* 2719, Springer, pp. 133–144, doi:10.1007/3-540-45061-0_12. Available at https://doi.org/10.1007/3-540-45061-0_12.

[5] Nadia Busi, Maurizio Gabbrielli & Gianluigi Zavattaro (2004): *Comparing Recursion, Replication, and Iteration in Process Calculi*. In Josep Díaz, Juhani Karhumäki, Arto Lepistö & Donald Sannella, editors: *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings*, *Lecture Notes in Computer Science* 3142, Springer, pp. 307–319, doi:10.1007/978-3-540-27836-8_28. Available at https://doi.org/10.1007/978-3-540-27836-8_28.

[6] Nadia Busi, Maurizio Gabbrielli & Gianluigi Zavattaro (2009): *On the expressive power of recursion, replication and iteration in process calculi*. *Math. Struct. Comput. Sci.* 19(6), pp. 1191–1222, doi:10.1017/S096012950999017X. Available at https://doi.org/10.1017/S096012950999017X.

[7] Luís Caires (2025): *Thread and Memory-Safe Programming with CLASS*. In Farzaneh Derakhshan & Jan Hoffmann, editors: *Proceedings 16th International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES@ETAPS 2025, Hamilton, Canada, 4th May 2025*, *EPTCS* 420, pp. 22–33, doi:10.4204/EPTCS.420.3. Available at https://doi.org/10.4204/EPTCS.420.3.

[8] Luís Caires & Frank Pfenning (2010): *Session Types as Intuitionistic Linear Propositions*. In Paul Gastin & François Laroussinie, editors: *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*, *Lecture Notes in Computer Science* 6269, Springer, pp. 222–236, doi:10.1007/978-3-642-15375-4_16. Available at https://doi.org/10.1007/978-3-642-15375-4_16.

[9] Ankush Das, Henry DeYoung, Andreia Mordido & Frank Pfenning (2021): *Nested Session Types*. In Nobuko Yoshida, editor: *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*, *Lecture Notes in Computer Science* 12648, Springer, pp. 178–206, doi:10.1007/978-3-030-72019-3_7. Available at https://doi.org/10.1007/978-3-030-72019-3_7.

[10] Ankush Das, Henry DeYoung, Andreia Mordido & Frank Pfenning (2022): *Nested Session Types*. *ACM Trans. Program. Lang. Syst.* 44(3), pp. 19:1–19:45, doi:10.1145/3539656. Available at https://doi.org/10.1145/3539656.

[11] Simon J. Gay, Diogo Poças & Vasco T. Vasconcelos (2022): *The Different Shades of Infinite Session Types*. In Patricia Bouyer & Lutz Schröder, editors: *Foundations of Software Science and Computation Structures - 25th International Conference, FOSSACS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*, *Lecture Notes in Computer Science* 13242, Springer, pp. 347–367, doi:10.1007/978-3-030-99253-8_18. Available at https://doi.org/10.1007/978-3-030-99253-8_18.

[12] Kohei Honda (1993): *Types for Dyadic Interaction*. In Eike Best, editor: *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, *Lecture Notes in Computer Science* 715, Springer, pp. 509–523, doi:10.1007/3-540-57208-2_35. Available at https://doi.org/10.1007/3-540-57208-2_35.

[13] Kohei Honda, Vasco Thudichum Vasconcelos & Makoto Kubo (1998): *Language Primitives and Type Discipline for Structured Communication-Based Programming*. In Chris Hankin, editor: *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, *Lecture Notes in Computer Science* 1381, Springer, pp. 122–138, doi:10.1007/BFB0053567. Available at `https://doi.org/10.1007/BFb0053567`.

[14] Dimitrios Kouzapas, Jorge A. Pérez & Nobuko Yoshida (2016): *On the Relative Expressiveness of Higher-Order Session Processes*. In Peter Thiemann, editor: *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, *Lecture Notes in Computer Science* 9632, Springer, pp. 446–475, doi:10.1007/978-3-662-49498-1_18. Available at `https://doi.org/10.1007/978-3-662-49498-1_18`.

[15] Dimitrios Kouzapas, Jorge A. Pérez & Nobuko Yoshida (2019): *On the relative expressiveness of higher-order session processes*. *Inf. Comput.* 268, doi:10.1016/J.IC.2019.06.002. Available at `https://doi.org/10.1016/j.ic.2019.06.002`.

[16] Ivan Lanese, Jorge A. Pérez, Davide Sangiorgi & Alan Schmitt (2008): *On the Expressiveness and Decidability of Higher-Order Process Calculi*. In: *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*, IEEE Computer Society, pp. 145–155, doi:10.1109/LICS.2008.8. Available at `https://doi.org/10.1109/LICS.2008.8`.

[17] Ivan Lanese, Jorge A. Pérez, Davide Sangiorgi & Alan Schmitt (2011): *On the expressiveness and decidability of higher-order process calculi*. *Inf. Comput.* 209(2), pp. 198–226, doi:10.1016/J.IC.2010.10.001. Available at `https://doi.org/10.1016/j.ic.2010.10.001`.

[18] Marvin L. Minsky (1967): *Computation: finite and infinite machines*. Prentice-Hall, Inc., USA.

[19] Pedro Rocha & Luís Caires (2023): *Safe Session-Based Concurrency with Shared Linear State*. In Thomas Wies, editor: *Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings*, *Lecture Notes in Computer Science* 13990, Springer, pp. 421–450, doi:10.1007/978-3-031-30044-8_16. Available at `https://doi.org/10.1007/978-3-031-30044-8_16`.

[20] Gil Silva, Andreia Mordido & Vasco T. Vasconcelos (2023): *Subtyping Context-Free Session Types*. In Guillermo A. Pérez & Jean-François Raskin, editors: *34th International Conference on Concurrency Theory, CONCUR 2023, Antwerp, Belgium, September 18-23, 2023*, *LIPIcs* 279, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 11:1–11:19, doi:10.4230/LIPICS.CONCUR.2023.11. Available at `https://doi.org/10.4230/LIPIcs.CONCUR.2023.11`.

[21] Peter Thiemann & Vasco T. Vasconcelos (2016): *Context-free session types*. In Jacques Garrigue, Gabriele Keller & Eijiro Sumii, editors: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, ACM, pp. 462–475, doi:10.1145/2951913.2951926. Available at `https://doi.org/10.1145/2951913.2951926`.

[22] Sjors Wortelboer (2026): *Nested Session Types in Turing Complete Calculi*. Master's thesis, University of Groningen. Available at `https://jperez.nl/files/MScThesis-Wortelboer.pdf`. Accessed: 2026-06-28.

A Additional Material for Section 2

A.1 Type rules ASP (Definition 3)

Type rules for processes P:

$$\frac{(k \in L) \qquad \mathcal{V};\Delta \vdash P :: (x : A_k)}{\mathcal{V};\Delta \vdash x.\mathtt{k};\ P :: (x : \oplus\{\mathtt{l} \Rightarrow A_l\}_{l\in L})}\ \oplus R \qquad \frac{(\forall l \in L) \qquad \mathcal{V};\Delta, x : A_l \vdash Q_l :: (z : C)}{\mathcal{V};\Delta, x : \oplus\{\mathtt{l} : A_l\}_{l\in L} \vdash \mathtt{case}\ x\ (\mathtt{l} \Rightarrow Q_l)_{l\in L} :: (z : C)}\ \oplus L$$

$$\frac{(\forall l \in L) \qquad \mathcal{V};\Delta \vdash P_l :: (x : A_l)}{\mathcal{V};\Delta \vdash \mathtt{case}\ x\ (\mathtt{l} \Rightarrow Q_l)_{l\in L} :: (x : \&\{\mathtt{l} \Rightarrow A_l\}_{l\in L})}\ \&R$$

$$\frac{(k \in L) \qquad \mathcal{V};\Delta \vdash P :: (x : A_k)}{\mathcal{V};\Delta, x : \&\{\mathtt{l} \Rightarrow A_l\}_{l\in L} \vdash x.\mathtt{k};\ Q :: (z : C)}\ \&L \qquad \frac{\cdot;\mathcal{V} \vdash A \equiv A' \qquad \mathcal{V};\Delta \vdash P :: (x : B)}{\mathcal{V};\Delta, y : A \vdash \mathtt{send}\ x\ y; P :: (x : A' \otimes B)}\ \otimes R$$

$$\frac{\mathcal{V};\Delta, y : A, x : B \vdash Q :: (z : C)}{\mathcal{V};\Delta, x : A \otimes B \vdash y \leftarrow \mathtt{recv}\ x; Q :: (z : C)}\ \otimes L \qquad \frac{\mathcal{V};\Delta, y : A \vdash P :: (x : B)}{\mathcal{V};\Delta \vdash y \leftarrow \mathtt{recv}\ x; P :: (x : A \multimap B)}\ \multimap R$$

$$\frac{\cdot;\mathcal{V} \vdash A \equiv A' \qquad \mathcal{V};\Delta, x : B \vdash Q :: (z : C)}{\mathcal{V};\Delta, x : A' \multimap B, y : A \vdash \mathtt{send}\ x\ y; Q :: (z : C)}\ \multimap L \qquad \frac{}{\cdot;\cdot \vdash \mathtt{close}\ x :: (x : \mathbf{1})}\ \mathbf{1}R$$

$$\frac{\mathcal{V};\Delta \vdash Q :: (z : C)}{\mathcal{V};\Delta, x : \mathbf{1} \vdash \mathtt{wait}\ x; Q :: (z : C)}\ \mathbf{1}L \qquad \frac{\cdot;\mathcal{V} \vdash A \equiv A'}{\mathcal{V}; y : A \vdash x \leftrightarrow y :: (x : A')}\ \mathrm{id}$$

$$\frac{\overline{y' : B'} \vdash f[\overline{\alpha}] = P_f :: (x' : B) \in \Sigma \qquad \cdot;\mathcal{V} \vdash \Delta' \equiv \overline{y : B'[\theta]} \qquad \mathcal{V};\Delta, x : B[\theta] \vdash Q :: (z : C)}{\mathcal{V};\Delta,\Delta' \vdash x \leftarrow f[\theta]\ \overline{y}; Q :: (z : C)}\ \mathrm{def}$$

Type rules for messages M:

$$\frac{(k \in L)}{\cdot; x' : A_k \vdash x.\mathtt{k};\ x \leftrightarrow x' :: (x : \oplus\{\mathtt{l} : A_l\}_{l\in L})}\ \oplus M \qquad \frac{(k \in L)}{\cdot; x : \&\{\mathtt{l} : A_l\}_{l\in L} \vdash x.\mathtt{k};\ x' \leftrightarrow x :: (x' : A_k)}\ \&M$$

$$\frac{\cdot;\cdot \vdash A \equiv A'}{\cdot; y : A, x' : B \vdash \mathtt{send}\ x\ y; x \leftrightarrow x' :: (x : A' \otimes B)}\ \otimes M$$

$$\frac{\cdot;\cdot \vdash A \equiv A'}{\cdot; x : A' \multimap B, y : A \vdash \mathtt{send}\ x\ y; x' \leftrightarrow x :: (x' : B)}\ \multimap M \qquad \frac{}{\cdot;\cdot \vdash \mathtt{close}\ x :: (x : \mathbf{1})}\ \mathbf{1}M$$

A.2 Reduction rules for ASP (Definition 5)

$$\mathtt{proc}(x, x.\mathtt{k};\ P) \longrightarrow \mathtt{proc}(x', P[x'/x]), \mathtt{msg}(x, x.\mathtt{k};\ x \leftrightarrow x') \qquad (\oplus S)$$

$$\mathtt{msg}(x, x.\mathtt{k};\ x \leftrightarrow x'), \mathtt{proc}(y, \mathtt{case}\ x\ (\mathtt{l} \Rightarrow Q_l)) \longrightarrow \mathtt{proc}(y, Q_k[x'/x]) \qquad (\oplus C)$$

$$\mathtt{proc}(d, x.\mathtt{k};\ P) \longrightarrow \mathtt{msg}(x, x.\mathtt{k};\ x \leftrightarrow x'), \mathtt{proc}(d, P[x'/x]) \qquad (\&S)$$

$$\mathtt{proc}(x, \mathtt{case}\ x\ (\mathtt{l} \Rightarrow Q_l)), \mathtt{msg}(x, x.\mathtt{k};\ x \leftrightarrow x') \longrightarrow \mathtt{proc}(x', Q_k[x'/x]) \qquad (\&C)$$

$$\texttt{proc}(x, \texttt{send}\ x\ w; P) \longrightarrow \texttt{proc}(x', P[x'/x]), \texttt{msg}(x, \texttt{send}\ x\ w; x \leftrightarrow x') \qquad (\otimes S)$$
$$\texttt{msg}(x, \texttt{send}\ x\ w; x \leftrightarrow x'), \texttt{proc}(y, w' \leftarrow \texttt{recv}\ x; Q) \longrightarrow \texttt{proc}(y, Q[x'/x, w/w']) \qquad (\otimes C)$$

$$\texttt{proc}(y, \texttt{send}\ x\ w; P) \longrightarrow \texttt{msg}(x, \texttt{send}\ x\ w; x \leftrightarrow x'), \texttt{proc}(y, P[x'/x]) \qquad (\multimap S)$$
$$\texttt{proc}(x, w' \leftarrow \texttt{recv}\ x; Q), \texttt{msg}(x, \texttt{send}\ x\ w; x \leftrightarrow x') \longrightarrow \texttt{proc}(x', Q[x'/x, w/w']) \qquad (\multimap C)$$

$$\texttt{proc}(x, \texttt{close}\ x) \longrightarrow \texttt{msg}(x, \texttt{close}\ x) \qquad (\mathbf{1}S)$$
$$\texttt{msg}(x, \texttt{close}\ x), \texttt{proc}(y, \texttt{wait}\ x; P) \longrightarrow \texttt{proc}(y, P) \qquad (\mathbf{1}C)$$

$$\texttt{msg}(d, M), \texttt{proc}(c, c \leftrightarrow d) \longrightarrow \texttt{msg}(c, M[c/d]) \qquad (\mathrm{id}^{+}\mathrm{C})$$
$$\texttt{proc}(c, c \leftrightarrow d), \texttt{msg}(e, M_c) \longrightarrow \texttt{msg}(e, M_c[d/c]) \qquad (\mathrm{id}^{-}\mathrm{C})$$

$$\texttt{proc}(d, Q), \texttt{proc}(c, c \leftrightarrow d) \longrightarrow \texttt{proc}(c, Q[c/d]) \qquad (\mathrm{id}^{+}\mathrm{S})$$
$$\texttt{proc}(c, c \leftrightarrow d), \texttt{proc}(e, Q_c) \longrightarrow \texttt{proc}(e, Q_c[d/c]) \qquad (\mathrm{id}^{-}\mathrm{S})$$

$$\texttt{proc}(z, x \leftarrow f[\theta]\ \overline{d}; Q) \longrightarrow \texttt{proc}(a, P_f[a/x, \overline{d}/\overline{y}, \theta]), \texttt{proc}(z, Q[a/x]) \qquad (\mathrm{defC})$$

B Additional Material for Section 3

B.1 Constituent process definitions (Definition 12)

$$\begin{aligned}
N^{0,0} = \texttt{case}\ z\ (&\texttt{exp}_0 \Rightarrow c \leftarrow t^{0,0}[]; z \leftarrow t^{1,0}[]\ c\\
&\texttt{exp}_1 \Rightarrow c \leftarrow t^{0,0}[]; z \leftarrow t^{0,1}[]\ c\\
&\texttt{dup} \Rightarrow c \leftarrow t^{0,0}[]; \texttt{send}\ z\ c; z \leftarrow t^{0,0}[]\\
&\texttt{ter} \Rightarrow \texttt{close}\ z)\\
N^{1,0} = \texttt{case}\ z\ (&\texttt{ins}_0 \Rightarrow r_0 \leftarrow \texttt{recv}\ z; r_1 \leftarrow \texttt{recv}\ z; y \leftarrow \texttt{recv}\ z;\\
&\qquad \texttt{case}\ z\ (\texttt{i} \Rightarrow z \leftarrow p_i^{0,0}[]\ r_0\ r_1\ a\ y)_{i \in LM}\\
&\texttt{exp}_0 \Rightarrow a.\texttt{exp}_0;\ z \leftarrow t^{s,0}[0[]]\ a\\
&\texttt{exp}_1 \Rightarrow a.\texttt{dup};\ b \leftarrow \texttt{recv}\ a; a.\texttt{exp}_1;\ b.\texttt{exp}_0;\ z \leftarrow t^{1,1}[]\ a\ b\\
&\texttt{dup} \Rightarrow a.\texttt{dup};\ a' \leftarrow \texttt{recv}\ a; c \leftarrow t^{1,0}[]\ a; \texttt{send}\ z\ c; z \leftarrow t^{1,0}[]\ a'\\
&\texttt{ter} \Rightarrow a.1;\ \texttt{wait}\ a; \texttt{close}\ z)\\
N^{s,0} = \texttt{case}\ z\ (&\texttt{ins}_0 \Rightarrow r_0 \leftarrow \texttt{recv}\ z; r_1 \leftarrow \texttt{recv}\ z; y \leftarrow \texttt{recv}\ z;\\
&\qquad \texttt{case}\ z\ (\texttt{i} \Rightarrow z \leftarrow p_i^{s,0}[\alpha]\ r_0\ r_1\ a\ y)_{i \in LM}\\
&\texttt{exp}_0 \Rightarrow a.\texttt{exp}_0;\ z \leftarrow t^{s,0}[S[\alpha]]\ a\\
&\texttt{exp}_1 \Rightarrow a.\texttt{dup};\ b \leftarrow \texttt{recv}\ a; a.\texttt{exp}_1;\ b.\texttt{exp}_0;\ z \leftarrow t^{s,1}[\alpha]\ a\ b\\
&\texttt{dup} \Rightarrow a.\texttt{dup};\ b \leftarrow \texttt{recv}\ a; c \leftarrow t^{s,0}[\alpha]\ a; \texttt{send}\ z\ c; z \leftarrow t^{s,0}[\alpha]\ b\\
&\texttt{ter} \Rightarrow a.\texttt{ter};\ \texttt{wait}\ a; \texttt{close}\ z)
\end{aligned}$$

$$
\begin{aligned}
N^{0,1} = \ & \mathtt{case}\ z\ (\mathtt{ins}_1 \Rightarrow r_0 \leftarrow \mathtt{recv}\ z; r_1 \leftarrow \mathtt{recv}\ z; y \leftarrow \mathtt{recv}\ z; \\
& \qquad \mathtt{case}\ z\ (\mathtt{i} \Rightarrow z \leftarrow p_i^{0,0}[]\ r_0\ r_1\ b\ y)_{i \in LM} \\
& \quad \mathtt{exp}_0 \Rightarrow b.\mathtt{dup};\ a \leftarrow \mathtt{recv}\ b; a.\mathtt{exp}_1;\ b.\mathtt{exp}_0;\ z \leftarrow t^{1,1}[]\ a\ b \\
& \quad \mathtt{exp}_1 \Rightarrow b.\mathtt{exp}_1;\ z \leftarrow t^{0,s}[0[]]\ b \\
& \quad \mathtt{dup} \Rightarrow a.\mathtt{dup};\ a' \leftarrow \mathtt{recv}\ a; c \leftarrow t^{0,1}[]\ a; \mathtt{send}\ z\ c; z \leftarrow t^{0,1}[]\ a' \\
& \quad \mathtt{ter} \Rightarrow b.1;\ \mathtt{wait}\ b; \mathtt{close}\ z) \\
N^{0,s} = \ & \mathtt{case}\ z\ (\mathtt{ins}_1 \Rightarrow r_0 \leftarrow \mathtt{recv}\ z; r_1 \leftarrow \mathtt{recv}\ z; y \leftarrow \mathtt{recv}\ z; \\
& \qquad \mathtt{case}\ z\ (\mathtt{i} \Rightarrow z \leftarrow p_i^{s,0}[\alpha]\ r_0\ r_1\ a\ y)_{i \in LM} \\
& \quad \mathtt{exp}_0 \Rightarrow b.\mathtt{dup};\ a \leftarrow \mathtt{recv}\ b; b.\mathtt{exp}_1;\ a.\mathtt{exp}_0;\ z \leftarrow t^{1,s}[\beta]\ b\ a \\
& \quad \mathtt{exp}_1 \Rightarrow b.\mathtt{exp}_1;\ z \leftarrow t^{0,s}[S[\beta]]\ a \\
& \quad \mathtt{dup} \Rightarrow b.\mathtt{dup};\ a \leftarrow \mathtt{recv}\ b; c \leftarrow t^{0,s}[\beta]\ b; \mathtt{send}\ z\ c; z \leftarrow t^{0,s}[\beta]\ a \\
& \quad \mathtt{ter} \Rightarrow b.\mathtt{ter};\ \mathtt{wait}\ b; \mathtt{close}\ z) \\
N^{1,1} = \ & \mathtt{case}\ z\ (\mathtt{ins}_0 \Rightarrow b.\mathtt{ter};\ \mathtt{wait}\ b; r_0 \leftarrow \mathtt{recv}\ z; r_1 \leftarrow \mathtt{recv}\ z; y \leftarrow \mathtt{recv}\ z; \\
& \qquad \mathtt{case}\ z\ (\mathtt{i} \Rightarrow z \leftarrow p_i^{0,s}[\alpha]\ r_0\ r_1\ a\ y)_{i \in LM} \\
& \quad \mathtt{ins}_1 \Rightarrow a.\mathtt{ter};\ \mathtt{wait}\ a; r_0 \leftarrow \mathtt{recv}\ z; r_1 \leftarrow \mathtt{recv}\ z; y \leftarrow \mathtt{recv}\ z; \\
& \qquad \mathtt{case}\ z\ (\mathtt{i} \Rightarrow z \leftarrow p_i^{s,0}[\alpha]\ r_0\ r_1\ b\ y)_{i \in LM} \\
& \quad \mathtt{exp}_0 \Rightarrow a.\mathtt{exp}_0;\ b.\mathtt{exp}_0;\ z \leftarrow t^{s,1}[0[]]\ a\ b \\
& \quad \mathtt{exp}_1 \Rightarrow a.\mathtt{exp}_1;\ b.\mathtt{exp}_1;\ z \leftarrow t^{1,s}[0[]]\ a\ b \\
& \quad \mathtt{ter} \Rightarrow a.\mathtt{ter};\ b.\mathtt{ter};\ \mathtt{wait}\ a; \mathtt{wait}\ b; \mathtt{close}\ z) \\
N^{s,1} = \ & \mathtt{case}\ z\ (\mathtt{ins}_0 \Rightarrow b.\mathtt{ter};\ \mathtt{wait}\ b; r_0 \leftarrow \mathtt{recv}\ z; r_1 \leftarrow \mathtt{recv}\ z; y \leftarrow \mathtt{recv}\ z; \\
& \qquad \mathtt{case}\ z\ (\mathtt{i} \Rightarrow z \leftarrow p_i^{s,s}[\alpha]\ r_0\ r_1\ a\ y)_{i \in LM} \\
& \quad \mathtt{ins}_1 \Rightarrow a.\mathtt{ter};\ \mathtt{wait}\ a; r_0 \leftarrow \mathtt{recv}\ z; r_1 \leftarrow \mathtt{recv}\ z; y \leftarrow \mathtt{recv}\ z; \\
& \qquad \mathtt{case}\ z\ (\mathtt{i} \Rightarrow z \leftarrow p_i^{s,0}[\alpha]\ r_0\ r_1\ b\ y)_{i \in LM} \\
& \quad \mathtt{exp}_0 \Rightarrow a.\mathtt{exp}_0;\ b.\mathtt{exp}_0;\ z \leftarrow t^{s,1}[S[\alpha]]\ a\ b \\
& \quad \mathtt{exp}_1 \Rightarrow a.\mathtt{exp}_1;\ b.\mathtt{exp}_1;\ z \leftarrow t^{s,s}[\alpha, 0[]]\ a\ b \\
& \quad \mathtt{ter} \Rightarrow a.\mathtt{ter};\ b.\mathtt{ter};\ \mathtt{wait}\ a; \mathtt{wait}\ b; \mathtt{close}\ z) \\
N^{1,s} = \ & \mathtt{case}\ z\ (\mathtt{ins}_0 \Rightarrow b.\mathtt{ter};\ \mathtt{wait}\ b; r_0 \leftarrow \mathtt{recv}\ z; r_1 \leftarrow \mathtt{recv}\ z; y \leftarrow \mathtt{recv}\ z; \\
& \qquad \mathtt{case}\ z\ (\mathtt{i} \Rightarrow z \leftarrow p_i^{0,s}[\beta]\ r_0\ r_1\ a\ y)_{i \in LM} \\
& \quad \mathtt{ins}_1 \Rightarrow a.\mathtt{ter};\ \mathtt{wait}\ a; r_0 \leftarrow \mathtt{recv}\ z; r_1 \leftarrow \mathtt{recv}\ z; y \leftarrow \mathtt{recv}\ z; \\
& \qquad \mathtt{case}\ z\ (\mathtt{i} \Rightarrow z \leftarrow p_i^{s,s}[\beta]\ r_0\ r_1\ b\ y)_{i \in LM} \\
& \quad \mathtt{exp}_0 \Rightarrow a.\mathtt{exp}_0;\ b.\mathtt{exp}_0;\ z \leftarrow t^{s,s}[0[], \beta]\ a\ b \\
& \quad \mathtt{exp}_1 \Rightarrow a.\mathtt{exp}_1;\ b.\mathtt{exp}_1;\ z \leftarrow t^{1,s}[S[\beta]]\ a\ b \\
& \quad \mathtt{ter} \Rightarrow a.\mathtt{ter};\ b.\mathtt{ter};\ \mathtt{wait}\ a; \mathtt{wait}\ b; \mathtt{close}\ z)
\end{aligned}
$$

$$
\begin{aligned}
N^{s,s} = \mathtt{case}\ z\ (&\mathtt{ins}_0 \Rightarrow b.\mathtt{ter};\ \mathtt{wait}\ b; r_0 \leftarrow \mathtt{recv}\ z; r_1 \leftarrow \mathtt{recv}\ z; y \leftarrow \mathtt{recv}\ z;\\
&\quad \mathtt{case}\ z\ (\mathtt{i} \Rightarrow z \leftarrow p_i^{s,s}[\alpha,\beta]\ r_0\ r_1\ a\ y)_{i\in L\mathcal{M}}\\
&\mathtt{ins}_1 \Rightarrow a.\mathtt{ter};\ \mathtt{wait}\ a; r_0 \leftarrow \mathtt{recv}\ z; r_1 \leftarrow \mathtt{recv}\ z; y \leftarrow \mathtt{recv}\ z;\\
&\quad \mathtt{case}\ z\ (\mathtt{i} \Rightarrow z \leftarrow p_i^{s,s}[\alpha,\beta]\ r_0\ r_1\ b\ y)_{i\in L\mathcal{M}}\\
&\mathtt{exp}_0 \Rightarrow a.\mathtt{exp}_0;\ b.\mathtt{exp}_0;\ z \leftarrow t^{s,s}[S[\alpha],\beta]\ a\ b\\
&\mathtt{exp}_1 \Rightarrow a.\mathtt{exp}_1;\ b.\mathtt{exp}_1;\ z \leftarrow t^{s,s}[\alpha,S[\beta]]\ a\ b\\
&\mathtt{ter} \Rightarrow a.\mathtt{ter};\ b.\mathtt{ter};\ \mathtt{wait}\ a; \mathtt{wait}\ b; \mathtt{close}\ z)
\end{aligned}
$$