CIRCULAR PROOFS AS SESSION-TYPED PROCESSES: A LOCAL VALIDITY CONDITION

FARZANEH DERAKHSHAN AND FRANK PFENNING

Philosophy Department, Carnegie Mellon University, Pittsburgh, PA, 15213 USA
e-mail address: fderakhs@andrew.cmu.edu

Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, 15213 USA
e-mail address: fp@cs.cmu.edu

Abstract. Proof theory provides a foundation for studying and reasoning about programming languages, most directly based on the well-known Curry-Howard isomorphism between intuitionistic logic and the typed lambda-calculus. More recently, a correspondence between intuitionistic linear logic and the session-typed pi-calculus has been discovered. In this paper, we establish an extension of the latter correspondence for a fragment of substructural logic with least and greatest fixed points. We describe the computational interpretation of the resulting infinitary proof system as session-typed processes, and provide an effectively decidable local criterion to recognize mutually recursive processes corresponding to valid circular proofs as introduced by Fortier and Santocanale. We show that our algorithm imposes a stricter requirement than Fortier and Santocanale's guard condition, but is local and compositional and therefore more suitable as the basis for a programming language.

Introduction

Proof theory provides a solid ground for studying and reasoning about programming languages. This logical foundation is mostly based on the well-known Curry-Howard isomorphism [How69] that establishes a correspondence between natural deduction and the typed λ-calculus by mapping propositions to types, proofs to well-typed programs, and proof reduction to computation. More recently, Caires et al. [CP10, CPT16] introduced a correspondence between intuitionistic linear logic [GL87] and the session-typed π-calculus that relates linear propositions to session types, proofs in the sequent calculus to concurrent processes, and cut reduction to computation. In this paper, we expand the latter for a fragment of intuitionistic linear logic called *subsingleton logic* in which the antecedent of each sequent consists of at most one formula. We consider the sequent calculus of subsingleton logic with least and greatest fixed points added to the underlying type system and their corresponding rules added to the calculus [DP16, DeY19]. We closely follow Fortier and Santocanale's [FS13] development in singleton logic, where the antecedent consists of exactly one formula.

Fortier and Santoconale [FS13, San02b] extend the sequent calculus for singleton logic with rules for least and greatest fixed points. A naive extension, however, loses the cut elimination property (even when allowing infinite proofs) so they call derivations *pre-proofs*. *Circular pre-proofs* are

Key words and phrases: Circular proofs, session types, subsingleton logic, fixed points, progress, termination, linear logic, substructural logic.

distinguished as a subset of derivations which are *regular* in the sense that they can be represented as finite trees with loops. Fortier and Santocanale then impose a validity condition (which we call the *FS guard condition*) on pre-proofs to single out a class of pre-proofs that satisfy cut elimination. Moreover, they provide a cut elimination algorithm and show that it locally terminates on derivations that satisfy the guard condition. In addition, Santocanale and Fortier [FS13, San02b, San02a] introduced a categorical semantics for interpreting cut elimination in singleton logic.

In a related line of research, Baelde et al. [BDS16, Bae12] add least and greatest fixed points to the sequent calculus for the multiplicative additive fragment of linear logic (*MALL*) that results in losing the cut elimination property. They also introduced a validity condition to distinguish circular proofs from infinite pre-proofs in *MALL*. Using Büchi automata, Doumane [Dou17] showed that the validity condition for identifying circular proofs in *MALL* with fixed points is PSPACE decidable. Nollet et al. [NST18] introduced a polynomial time algorithm for locally identifying a stricter version of Baelde's condition in *MALL* with fixed points.

In this paper, we study (mutually) recursive session-typed processes and their correspondence with circular pre-proofs in subsingleton logic with fixed points. We introduce an algorithm to check a stricter version of the FS guard condition. Our algorithm is local in the sense that we check validity of each process separately, and it is stricter in the sense that it accepts a proper subset of the proofs recognized by the FS guard condition. We further introduce a synchronous computational semantics of cut reduction in subsingleton logic with fixed points in the context of session types, based on a key step in Fortier and Santocanale's cut elimination algorithm which is compatible with prior operational interpretations of session-typed programming languages [TCP13]. We show preservation and a strong version of the progress property that ensures that each valid process communicates along its left or right interface in a finite number of steps. A key aspect of our type system is that validity is a compositional property (as we generally expect from type systems) so that the composition of valid programs defined over the same signature are also valid and therefore also satisfy strong progress.

In the session type system, a singleton judgment $A \vdash B$ is annotated as $x : A \vdash \mathtt{P} :: (y : B)$ which is understood as: process $\mathtt{P}$ uses a service of type A offered along channel x by a process on its left and provides a service of type B along channel y to a process on its right [DP16]. The left and right interfaces of a process in the session type system inherit the symmetry of the left and right rules in the sequent calculus. Each process interacts with other processes along its pair of left and right interfaces that are corresponding to the left and right sides of a sequent. For example, two processes $\mathtt{P}$ and $\mathtt{Q}$ with the typing $x : A \vdash \mathtt{P} :: (y : B)$ and $y : B \vdash \mathtt{Q} :: (z : C)$ can be composed so they interact with each other using channel y. Process $\mathtt{P}$ provides a service of type B and offers it along channel y and process $\mathtt{Q}$ uses this service to provide its own service of type C. This interaction along channel y can be of two forms: (i) process $\mathtt{P}$ sends a message to the right, and process $\mathtt{Q}$ receives it from the left, or (ii) process $\mathtt{Q}$ sends a message to the left and process $\mathtt{P}$ receives it from the right. In the first case, the session type B is a *positive* type, and in the second case it is a *negative* type. Least fixed points have a positive polarity while greatest fixed points are negative [LM16]. As we will see in Sections 5 and 6, due to the interactive nature of computation some types that would be considered "empty" (that is, have no closed values) may still be of interest here.

DeYoung and Pfenning [DP16, Pfe16] provide a representation of Turing machines in the session-typed rule system of subsingleton logic with general equirecursive types. This shows that cut reduction on circular pre-proofs in subsingleton logic with equirecursive types has the computational power of Turing machines. Using this encoding on isorecursive types, we show that recognizing all programs that satisfy a strong progress property is undecidable, since this property can be encoded as termination of a Turing machine on a given input. However, with our algorithm, we can decide validity of a subset of Turing machines represented as session-typed processes in subsingleton logic with isorecursive fixed points.

In summary, the principal contribution of our paper is to extend the Curry-Howard interpretation of proofs in subsingleton logic as communicating processes to include least and greatest fixed points. A circular proof is represented as a collection of mutually recursive process definitions. We develop a compositional criterion for validity of such programs, which is local in the sense that each process definition can be checked independently. Local validity in turn implies a strong progress property on programs and cut elimination on the circular proofs they correspond to.

The structure of the remainder of the paper is as follows. In Section 1 we introduce subsingleton logic with fixed points, and then examine it in the context of session-typed communication (Section 2). We provide a process notation with a synchronous operational semantics in Section 3 and a range of examples in Section 4. We then develop a local validity condition through a sequence of refinements in Sections 5–8. We capture this condition on infinitary proofs in Section 9 and reduce it to a finitary algorithm in Section 10. We prove that local validity implies Fortier and Santocanale's guard condition (Section 11) and therefore cut elimination. In Section 12 we explore the computational consequences of this, including the strong progress property which states that every valid configuration of processes will either be empty or attempt to communicate along external channels after a finite number of steps. We conclude by illustrating some limitations of our algorithm (Section 13) and pointing to some additional related and future work (Section 14).

1. Subsingleton Logic with Fixed Points

Subsingleton logic is a fragment of intuitionistic linear logic [GL87, CCP03] in which the antecedent and succedent of each judgment consist of at most one proposition. This reduces consideration to the additive connectives and multiplicative units, because the left or right rules of other connectives would violate this restriction. The expressive power of pure subsingleton logic is rather limited, among other things due to the absence of the exponential $!A$. However, we can recover significant expressive power by adding least and greatest fixed points, which can be done without violating the subsingleton restriction. We think of subsingleton logic as a laboratory in which to study the properties and behaviors of least and greatest fixed points in their simplest nontrivial form, following the seminal work of Fortier and Santocanale [FS13].

The syntax of propositions then follows the grammar

$$A, B ::= A \oplus B \mid 0 \mid A \& B \mid \top \mid 1 \mid \bot \mid t$$

where t stands for propositions defined by least or greatest fixed points. Rather than including these directly as $\mu t. A$ and $\nu t. A$, we define them in a *signature* Σ which records some important additional information, namely their relative *priority*.

$$\Sigma ::= \cdot \mid \Sigma, t =^i_\mu A \mid \Sigma, t =^i_\nu A,$$

with the condition that if $t =^i_a A \in \Sigma$ and $t =^i_b B \in \Sigma$, then $a = b$.

For a fixed point t defined as $t =^i_a A$ in Σ, the subscript a is the *polarity* of t: if $a = \mu$, then t is a fixed point with *positive* polarity and if $a = \nu$, then it is of *negative* polarity. Finitely representable least fixed points (e.g., natural numbers and lists) can be represented in this system as defined type variables with positive polarity, while the potentially infinite greatest fixed points (e.g., streams and infinite depth trees) are represented as those with negative polarity.

The superscript i is the *priority* of t. Fortier and Santocanale interpreted the priority of fixed points in their system as the order in which the least and greatest fixed point equations are solved in

$$\frac{}{A \vdash A}\,\textsc{Id}_A \qquad \frac{\omega \vdash A \quad A \vdash \gamma}{\omega \vdash \gamma}\,\textsc{Cut}_A$$

$$\frac{\omega \vdash A}{\omega \vdash A \oplus B}\oplus R_1 \qquad \frac{\omega \vdash B}{\omega \vdash A \oplus B}\oplus R_2 \qquad \frac{A \vdash \gamma \quad B \vdash \gamma}{A \oplus B \vdash \gamma}\oplus L$$

$$\frac{\omega \vdash A \quad \omega \vdash B}{\omega \vdash A \& B}\& R \qquad \frac{A \vdash \gamma}{A \& B \vdash \gamma}\& L_1 \qquad \frac{B \vdash \gamma}{A \& B \vdash \gamma}\& L_2$$

$$\frac{}{\cdot \vdash 1}1R \qquad \frac{\cdot \vdash \gamma}{1 \vdash \gamma}1L \qquad \frac{\omega \vdash \cdot}{\omega \vdash \bot}\bot R \qquad \frac{}{\bot \vdash \cdot}\bot L$$

$$\frac{\omega \vdash A \quad t =_\mu A \in \Sigma}{\omega \vdash t}\mu R \quad \frac{A \vdash \gamma \quad t =_\mu A \in \Sigma}{t \vdash \gamma}\mu L \quad \frac{\omega \vdash A \quad t =_\nu A \in \Sigma}{\omega \vdash t}\nu R \quad \frac{A \vdash \gamma \quad t =_\nu A \in \Sigma}{t \vdash \gamma}\nu L$$

Figure 1: Sequent calculus for subsingleton logic with fixed points.

the semantics [FS13]. We use them syntactically as central information to determine local validity of circular proofs. We write $p(t) = i$ for the priority of t, and $\epsilon(i) = a$ for the polarity of type t with priority i. The condition on Σ ensures that ϵ is a well-defined function.

The basic judgment of the subsingleton sequent calculus has the form $\omega \vdash_\Sigma \gamma$, where ω and γ are either empty or a single proposition A and Σ is a signature. Since the signature never changes in the rules, we omit it from the turnstile symbol. The rules of subsingleton logic with fixed points are summarized in Figure 1, constituting a slight generalization of Fortier and Santocanale's. When the fixed points in the last row are included, this set of rules must be interpreted as *infinitary*, meaning that a judgment may have an infinite derivation in this system.

Even a cut-free derivation may be of infinite length since each defined type variable may be unfolded infinitely many times. Also, cut elimination no longer holds for the derivations after adding fixed point rules. What the rules define then are the so-called *(infinite) pre-proofs* which include infinite derivations that do not necessarily enjoy the cut elimination property. In particular, we are interested in *circular pre-proofs*. Circular pre-proofs are those infinite pre-proofs that can be illustrated as finite trees with loops [Dou17].

As an example, the following circular pre-proof defined on the signature $\mathsf{nat} =^1_\mu 1 \oplus \mathsf{nat}$ depicts an infinite pre-proof that consists of repetitive application of μR followed by $\oplus R$:

$$\cfrac{\cfrac{\cdot \vdash \mathsf{nat}}{\cdot \vdash 1 \oplus \mathsf{nat}}\oplus R}{\cdot \vdash \mathsf{nat}}\mu R$$

It turns out not to be valid. On the other hand, on the signature $\mathsf{conat} =^1_\nu 1 \,\&\, \mathsf{conat}$, we can define a circular pre-proof using greatest fixed points that is valid:

$$\cfrac{\cfrac{\cdot \vdash 1 \quad \cdot \vdash \mathsf{conat}}{\cdot \vdash 1 \& \ \mathsf{conat}}\& R}{\cdot \vdash \mathsf{conat}}\nu R$$

2. Fixed Points in the Context of Session Types

Session types [Hon93, HVK98] describe the communication behavior of interacting processes. *Binary session types*, where each channel has two endpoints, have been recognized as arising from linear logic (either in its intuitionistic [CP10, CPT16] or classical [Wad12] formulation) by a Curry-Howard interpretion of propositions as types, proofs as programs, and cut reduction as communication. In the context of programming, recursive session types have also been considered [TCP13, LM16], and they seem to fit smoothly, just as recursive types fit well into functional programming languages. However, they come at a price, since we abandon the Curry-Howard correspondence.

In this paper we show that this is not necessary: we can remain on a sound logical footing as long as we (a) refine general recursive session types into least and greatest fixed points, (b) are prepared to accept circular proofs, and (c) impose conditions under which recursively defined processes are valid. General (nonlinear) type theory has followed a similar path, isolating inductive and coinductive types with a variety of conditions to ensure validity of proofs. In the setting of subsingleton logic, however, we find many more symmetries than typically present in traditional type theories that appear to be naturally biased towards least fixed points and inductive reasoning.

Under the Curry-Howard interpretation a subsingleton judgment $A \vdash_\Sigma B$ is annotated as

$$x : A \vdash_\Sigma \mathtt{P} :: (y : B)$$

where x and y are two different channels and A and B are their corresponding session types. One can understand this judgment as: process $\mathtt{P}$ provides a service of type B along channel y while using channel x of type A, a service that is provided by another process along channel x [DP16]. We can form a chain of processes $\mathtt{P}_0, \mathtt{P}_1, \cdots, \mathtt{P}_n$ with the typing

$$\cdot \vdash \mathtt{P}_0 :: (x_0 : A_0), \quad x_0 : A_0 \vdash \mathtt{P}_1 :: (x_1 : A_1), \quad \cdots \quad x_{n-1} : A_{n-1} \vdash \mathtt{P}_n :: (x_n : A_n)$$

which we write as

$$\mathtt{P}_0 \mid_{x_0} \mathtt{P}_1 \mid_{x_1} \cdots \mid_{x_{n-1}} \mathtt{P}_n$$

in analogy with the notation for parallel composition for processes $P \mid Q$, although here it is not commutative. In such a chain, process $\mathtt{P}_{i+1}$ uses a service of type A_i provided by the process $\mathtt{P}_i$ along the channel x_i, and provides its own service of type A_{i+1} along the channel x_{i+1}. Process $\mathtt{P}_0$ provides a service of type A_0 along channel x_0 without using any services. So, a process in the session type system, instead of being reduced to a value as in the functional programming, interacts with both its left and right interfaces by sending and receiving messages. Processes $\mathtt{P}_i$ and $\mathtt{P}_{i+1}$, for example, communicate with each other along the channel x_i of type A_i: If process $\mathtt{P}_i$ sends a message along channel x_i to the right and process $\mathtt{P}_{i+1}$ receives it from the left (along the same channel), session type A_i is called a *positive* type. And if process $\mathtt{P}_{i+1}$ sends a message along channel x_i to the left and process $\mathtt{P}_i$ receives it from the right (along the same channel), session type A_i is called a *negative* type. In Section 4 we show in detail that this symmetric behavior of left and right session types results in a symmetric behaviour of least and greatest fixed point types.

In general, in a chain of processes, the leftmost type may not be empty. Also, strictly speaking, the names of the channels are redundant since every process has two distinguished ports: one to the left and one to the right, either one of which may be empty. Because of this, we may sometimes omit the channel name, but in the theory we present in this paper it is convenient to always refer to communication channels by unique names.

For programming examples, it is helpful to allow any finite number of alternatives for internal ($\oplus$) and external ($\&$) choice. Finitary choices can equally well interpreted as propositions, so this is not a departure from the proofs as programs interpretation.

Definition 1. We define *session types* with following grammar, where L ranges over finite sets of labels denoted by ℓ and k.

$$A ::= \oplus\{\ell : A_\ell\}_{\ell \in L} \mid \&\{\ell : A_\ell\}_{\ell \in L} \mid 1 \mid \bot \mid t$$

where t are type variables whose definition is given in a signature Σ as before. The binary disjunction and conjunction are defined as $A \oplus B = \oplus\{\pi_1 : A, \pi_2 : B\}$ and $A \& B = \&\{\pi_1 : A, \pi_2 : B\}$. Similarly, we define $0 = \oplus\{\}$ and $\top = \&\{\}$.

As a first programming-related example, consider natural numbers in unary form (nat) and infinite streams of natural numbers (stream).

Example 1 (Streams of natural numbers)**.**

$$\mathsf{nat} =^1_\mu \oplus\{z : 1, s : \mathsf{nat}\}$$
$$\mathsf{stream} =^2_\nu \&\{\mathit{head} : \mathsf{nat},\ \ \mathit{tail} : \mathsf{stream}\}$$

Σ, in this example, consists of two well-known, respectively, inductive and coinductive types: (i) the type of natural numbers (nat) built using two constructors for *zero* and *successor*, and (ii) the type of infinite streams (stream) defined using two destructors for *head* and *tail* of a stream. Here, priorities of nat and stream are, respectively, 1 and 2, understood as "nat *has higher priority than* stream". △

Example 2 (Binary numbers in standard form)**.** As another example consider the signature with two types with positive polarity and the same priority: std and pos. Here, std is the type of standard bit strings, i.e., bit strings terminated with \$ without any leading 0 bits, and pos is the type of positive standard bit strings, i.e., all standard bit strings except \$.

$$\mathsf{std} =^1_\mu \oplus\{\mathit{b0} : \mathsf{pos}, \mathit{b1} : \mathsf{std}, \$: 1\}$$
$$\mathsf{pos} =^1_\mu \oplus\{\mathit{b0} : \mathsf{pos}, \mathit{b1} : \mathsf{std}\}$$

△

Example 3 (Bits and cobits)**.**

$$\mathsf{bits} =^1_\mu \oplus\{\mathit{b0} : \mathsf{bits},\ \mathit{b1} : \mathsf{bits}\}$$
$$\mathsf{cobits} =^2_\nu \&\{\mathit{b0} : \mathsf{cobits},\ \mathit{b1} : \mathsf{cobits}\}$$

In the functional programming type system cobits is a greatest fixed point recognized as an infinite stream of bits, while bits is recognized as an empty type. However, in the session type system, we treat them in a symmetric way. bits is an infinite sequence of bits with positive polarity. And its dual type, cobits, is an infinite stream of bits with negative polarity. In Examples 7 and 8, in Section 5, we illustrate more the symmetry of these types by providing two recursive processes having them as their interfaces. Even though we can, for example, write transducers of type $\mathsf{bits} \vdash \mathsf{bits}$ inside the language, we cannot write a *valid* process of type $\cdot \vdash \mathsf{bits}$ that *produces* an infinite stream. △

3. A Synchronous Operational Semantics

The operational semantics for process expression under the proofs-as-programs interpretation of linear logic has been treated exhaustively elsewhere [CP10, CPT16, TCP13, Gri16]. We therefore only briefly sketch the operational semantics here. Communication is *synchronous*, which means both sender and receiver block until they synchronize. Asynchronous communication can be modeled using a process with just one output action followed by forwarding [GV10, DCPT12]. However, a significant difference to much prior work is that we treat types in an isorecursive way, that is, a message is sent to unfold the definition of a type t. This message is written as μ_t for a least fixed point and ν_t for a greatest fixed point. The language of process expressions and their operational semantics presented in this section is suitable for general isorecursive types, if those are desired in an implementation. The resulting language then satisfies a weaker progress property sometimes called *local progress* (see, for example, [CP10]).

Definition 2. Processes are defined as follows over the signature Σ:

$$
\begin{array}{llr}
P, Q ::= & y \leftarrow x & \textit{identity} \\
& \mid (x \leftarrow P_x; Q_x) & \textit{cut} \\
& \mid Lx.k; P \mid \textbf{case}\, Rx\ (\ell \Rightarrow Q_\ell)_{\ell \in L} & \&\{\ell : A_\ell\}_{\ell \in L} \\
& \mid Rx.k; P \mid \textbf{case}\, Lx\ (\ell \Rightarrow Q_\ell)_{\ell \in L} & \oplus\{\ell : A_\ell\}_{\ell \in L} \\
& \mid \textbf{wait}\, Lx \mid \textbf{close}\, Rx; Q & 1 \\
& \mid \textbf{close}\, Lx; Q \mid \textbf{wait}\, Rx & \bot \\
& \mid Rx.\mu_t; P \mid \textbf{case}\, Lx\ (\mu_t \Rightarrow P) & t =_\mu A \\
& \mid Lx.\nu_t; P \mid \textbf{case}\, Rx\ (\nu_t \Rightarrow P) & t =_\nu A \\
& \mid y \leftarrow X \leftarrow x &
\end{array}
$$

where $X, Y, \ldots$ are *process variables*, and $x, y, \ldots$ are channel names. The left and right session types a process interacts with are uniquely labelled with channel names:

$$x : A \vdash P :: (y : B).$$

We read this as

Process P uses channel x of type A and provides a service of type B along channel y.

However, since a process may not use any channel to provide a service along its right channel, e.g. $\cdot \vdash P :: (y : B)$, or it may not provide any service along its right channel , e.g.$x : A \vdash Q :: (\cdot)$, the labelling of processes is generalized to be of the form:

$$\bar{x} : \omega \vdash P :: (\bar{y} : \gamma),$$

where $\bar{x}$ ($\bar{y}$) is either empty or x (y), and ω (γ) is empty given that $\bar{x}$ ($\bar{y}$) is empty.

Process definitions are of the form $\bar{x} : \omega \vdash X = P_{\bar{x},\bar{y}} :: (\bar{y} : B)$ representing that variable X is defined as process P. A *program* $\mathcal{P}$ is defined as a pair $\langle V, S \rangle$, where V is a finite set of process definitions, and S is the *main* process variable. Figure 2 shows the logical rules annotated with processes in the context of session types. This set of rules inherits the full symmetry of its underlying sequent calculus. These typing rules interpret *pre-proofs*: As can be seen in the rule Def, the shown typing rules inherit the infinitary nature of deductions from the logical rules in Figure 1 and are therefore not directly useful for type checking. We obtain a finitary system to check *circular* pre-proofs by removing the first premise from the Def rule and checking each process definition in V separately, under the hypothesis that all process definitions are well-typed. In order to also enforce validity, the rules need to track additional information (see rules in Figures 3 (infinitary) and 4 (finitary) in Sections 9 and 10).

All processes we consider in this paper provide a service along their right channel so in the remainder of the paper we restrict the sequents to be of the form $\bar{x} : \omega \vdash P :: (y : A)$. We therefore do not need to consider the rules for type $\bot$ anymore, but the results of this paper can easily be generalized to the fully symmetric calculus.

$$\frac{}{x : A \vdash y \leftarrow x :: (y : A)}\textsc{Id} \qquad \frac{\bar{x} : \omega \vdash P_w :: (w : A) \quad w : A \vdash Q_w :: (\bar{y} : \gamma)}{\bar{x} : \omega \vdash (w \leftarrow P_w ; Q_w) :: (\bar{y} : \gamma)}\textsc{Cut}^w$$

$$\frac{\bar{x} : \omega \vdash P :: (y : A_k) \quad (k \in L)}{\bar{x} : \omega \vdash Ry.k ; P :: (y : \oplus\{\ell : A_\ell\}_{\ell \in L})}\oplus R \qquad \frac{\forall \ell \in L \quad x : A_\ell \vdash P_\ell :: (\bar{y} : \gamma)}{x : \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \mathbf{case}\, Lx\, (\ell \Rightarrow P_\ell) :: (\bar{y} : \gamma)}\oplus L$$

$$\frac{\bar{x} : \omega \vdash P_\ell :: (y : A_\ell) \quad \forall \ell \in L}{\bar{x} : \omega \vdash \mathbf{case}\, Ry\, (\ell \Rightarrow P_\ell) :: (y : \&\{\ell : A_\ell\}_{\ell \in L})}\& R \qquad \frac{k \in L \quad x : A_k \vdash P :: (\bar{y} : \gamma)}{x : \&\{\ell : A_\ell\}_{\ell \in L} \vdash Lx.k ; P :: (\bar{y} : \gamma)}\& L$$

$$\frac{}{\cdot \vdash \mathbf{close}\, Ry :: (y : 1)}1R \qquad \frac{. \vdash Q :: (\bar{y} : \gamma)}{x : 1 \vdash \mathbf{wait}\, Lx ; Q :: (\bar{y} : \gamma)}1L$$

$$\frac{\bar{x} : A \vdash Q :: \cdot}{\bar{x} : A \vdash \mathbf{wait}\, Ry ; Q :: (y : \bot)}\bot R \qquad \frac{}{x : \bot \vdash \mathbf{close}\, Lx :: \cdot}\bot L$$

$$\frac{\bar{x} : \omega \vdash P_y :: (y : A) \quad t =_\mu A}{\bar{x} : \omega \vdash Ry.\mu_t ; P_y :: (y : t)}\mu R \qquad \frac{x : A \vdash Q_x :: (\bar{y} : \gamma) \quad t =_\mu A}{x : t \vdash \mathbf{case}\, Lx\, (\mu_t \Rightarrow Q_x) :: (\bar{y} : \gamma)}\mu L$$

$$\frac{\bar{x} : \omega \vdash P_y :: (y : A) \quad t =_\nu A}{\bar{x} : \omega \vdash \mathbf{case}\, Ry\, (\nu_t \Rightarrow P_y) :: (y : t)}\nu R \qquad \frac{x : A \vdash Q_x :: (\bar{y} : \gamma) \quad t =_\nu A}{x : t \vdash Lx.\nu_t ; Q_x :: (\bar{y} : \gamma)}\nu L$$

$$\frac{\bar{x} : \omega \vdash P_{\bar{x},\bar{y}} :: (\bar{y} : \gamma) \quad \bar{u} : \omega \vdash X = P_{\bar{u},\bar{w}} :: (\bar{w} : \gamma) \in V}{\bar{x} : \omega \vdash \bar{y} \leftarrow X \leftarrow \bar{x} :: (\bar{y} : \gamma)}\textsc{Def}(X)$$

Figure 2: Process assignment for subsingleton logic with fixed points (infinitary).

The computational semantics is defined on configurations

$$P_0 \mid_{x_1} \cdots \mid_{x_n} P_n$$

where $\mid$ is associative and has unit $(\cdot)$ but is not commutative. The following transitions can be applied anywhere in a configuration:

$P \mid_x (y \leftarrow x) \mid_y Q$	$\mapsto$	$([z/x]P) \mid_z ([z/y]Q)$	(z fresh), forward
$(x \leftarrow P ; Q)$	$\mapsto$	$([z/x]P) \mid_z ([z/x]Q)$	(z fresh), spawn
$(Rx.k ; P) \mid_x (\mathbf{case}\, Lx\, (\ell \Rightarrow Q_\ell))$	$\mapsto$	$P \mid_x Q_k$	send label k right
$\mathbf{case}\, Rx\, (\ell \Rightarrow P_\ell) \mid_x (Lx.k ; Q)$	$\mapsto$	$P_k \mid_x Q$	send label k left
$\mathbf{close}\, Rx \mid_x (\mathbf{wait}\, Lx ; Q)$	$\mapsto$	Q	close channel right
$(\mathbf{wait}\, Rx ; P) \mid_x \mathbf{close}\, Lx$	$\mapsto$	P	close channel left
$(Rx.\mu_t ; P) \mid_x (\mathbf{case}\, Lx\, (\mu_t \Rightarrow Q))$	$\mapsto$	$P \mid_x Q$	send μ_t unfolding message right
$\mathbf{case}\, Rx\, (\nu_t \Rightarrow P) \mid_x (Lx.\nu_t ; Q)$	$\mapsto$	$P \mid_x Q$	send ν_t unfolding message left
$\cdots \mid_{\bar{x}} \bar{y} \leftarrow X \leftarrow \bar{x} \mid_{\bar{y}} \cdots$	$\mapsto$	$\cdots \mid_{\bar{x}} [\bar{y}/\bar{w}, \bar{x}/\bar{u}]P \mid_{\bar{y}} \cdots$	where $\bar{u} : \omega \vdash X = P :: (\bar{w} : \gamma)$

The forward rule removes process $y \leftarrow x$ from the configuration and replaces both channels x and y in the rest of configuration with a fresh channel z. The rule for $x \leftarrow P ; Q$ spawns process $[z/x]P$ and continues as $[z/x]Q$. To ensure uniqueness of channels, we need z to be a fresh channel. For internal choice, $Rx.k; P$ sends label k along channel x to the process on its right and continues as P. The process on the right, **case** $Lx\,(\ell \Rightarrow Q_\ell)$, receives the label k sent from the left along channel x, and chooses the kth alternative Q_k to continue with accordingly. The last transition rule unfolds the definition of a process variable X while instantiating the left and right channels $\bar{u}$ and $\bar{w}$ in the process definition with proper channel names, $\bar{x}$ and $\bar{y}$ respectively.

4. Examples of Session-Typed Processes

In this section we motivate our local validity algorithm using a few examples. By defining type variables in the signature and process variables in the program, we can generate (mutually) recursive processes which correspond to circular pre-proofs in the sequent calculus. In Examples 4-5, we provide such recursive processes along with explanation of their computational steps and their corresponding derivations.

Example 4.

$$\Sigma_1 := \mathsf{nat} =^1_\mu \oplus\{z : 1, s : \mathsf{nat}\}.$$

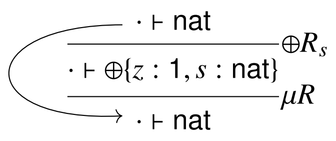
We define a process

$$\cdot \vdash \mathsf{Loop} :: (y : \mathsf{nat}),$$

where the variable Loop is defined as

$$\begin{array}{rlll}
y \leftarrow \mathsf{Loop} \leftarrow \cdot = & Ry.\mu_{nat}; & \textit{\% send } \mu_{nat} \textit{ to right} & \text{(i)}\\
& Ry.s; & \textit{\% send label s to right} & \text{(ii)}\\
& y \leftarrow \mathsf{Loop} \leftarrow \cdot & \textit{\% recursive call} & \text{(iii)}
\end{array}$$

$\mathcal{P}_1 := \langle\{\mathsf{Loop}\}, \mathsf{Loop}\rangle$ forms a program over the signature Σ_1. It (i) sends a *positive* fixed point unfolding message to the right, (ii) sends the label s, as another message corresponding to *successor*, to the right, (iii) calls itself and loops back to (i). The program runs forever, sending *successor* labels to the right, without receiving any fixed point unfolding messages from the left or the right. The process Loop corresponds to the following infinite derivation:


$\cdot \vdash \mathsf{nat}$
$\oplus R_s$
$\cdot \vdash \oplus\{z : 1, s : \mathsf{nat}\}$
μR
$\cdot \vdash \mathsf{nat}$

△

Example 5. Define process

$$x : \mathsf{nat} \vdash \mathtt{Block} :: (y : 1)$$

over the signature Σ_1 as

$$\begin{array}{llll}
y \leftarrow \mathtt{Block} \leftarrow x = & & & \\
\quad \textbf{case}\; Lx\; (\mu_{nat} \Rightarrow & & \textit{\% receive } \mu_{nat} \textit{ from left} & \text{(i)}\\
\quad\quad \textbf{case}\; Lx & & \textit{\% receive a label from left} & \text{(ii)}\\
\quad\quad\quad (\, z \Rightarrow \textbf{wait}\; Lx; & & \textit{\% wait and close x} & \text{(ii-a)}\\
\quad\quad\quad\quad \textbf{close}\; Ry & & \textit{\% close y} & \\
\quad\quad\quad \mid s \Rightarrow y \leftarrow \mathtt{Block} \leftarrow x)) & & \textit{\% recursive call} & \text{(ii-b)}
\end{array}$$

$\mathcal{P}_2 := \langle\{\mathtt{Block}\}, \mathtt{Block}\rangle$ forms a program over the signature Σ_1:
(i) $\mathtt{Block}$ **waits**, until it receives a *positive* fixed point unfolding message from the left, (ii) waits for

another message from the left to determine the path it will continue with:

(a) If the message is a z label, (ii-a) the program waits until a closing message is received from the left. Upon receiving that message, it closes the left and then the right channel.

(b) If the message is an s label, (ii-b) the program calls itself and loops back to (i).

Process `Block` corresponds to the following infinite derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot \vdash 1}\,1R}{1 \vdash 1}\,1L \qquad \mathsf{nat} \vdash 1}{\oplus\{z : 1, s : \mathsf{nat}\} \vdash 1}\,\oplus L}{\mathsf{nat} \vdash 1}\,\mu L$$

△

Derivations corresponding to both of these programs are cut-free. Also no internal loop takes place during their computation, in the sense that they both communicate with their left or right channels after finite number of steps. For process `Loop` this communication is restricted to sending infinitely many unfolding and successor messages to the right. Process `Block`, on the other hand, receives the same type of messages after finite number of steps as long as they are provided by a process on its left. Composing these two processes as in $x \leftarrow$ `Loop` $\leftarrow \cdot \mid y \leftarrow$ `Block` $\leftarrow x$ results in an internal loop: process `Loop` keeps providing unfolding and successor messages for process `Block` so that they both can continue the computation and call themselves recursively. Because of this internal loop, the composition is not acceptable: It never communicates with its left (empty channel) or right (channel y). The infinite derivation corresponding to the composition $x \leftarrow$ `Loop` $\leftarrow \cdot \mid y \leftarrow$ `Block` $\leftarrow x$ therefore should be rejected as invalid:

$$\dfrac{\dfrac{\dfrac{\cdot \vdash \mathsf{nat}}{\cdot \vdash \oplus\{z : 1, s : \mathsf{nat}\}}\,\oplus R_s}{\cdot \vdash \mathsf{nat}}\,\mu R \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\cdot \vdash 1}\,1R}{1 \vdash 1}\,1L \qquad \mathsf{nat} \vdash 1}{\oplus\{z : 1, s : \mathsf{nat}\} \vdash 1}\,\oplus L}{\mathsf{nat} \vdash 1}\,\mu L}{\cdot \vdash 1}\,\textsc{Cut}_{nat}$$

The cut elimination algorithm introduced by Fortier and Santocanale uses a reduction function Treat that may never halt. They proved that for derivations satisfying the guard condition Treat is locally terminating since it always halts on guarded proofs [FS13]. The above derivation is an example of one that does not satisfy the FS guard condition and the cut elimination algorithm does not locally terminate on it.

The *progress property* for a configuration of processes ensures that during its computation it either (i) takes a step, or (ii) is empty, or (iii) communicates along its left or right channel. Without (mutually) recursive processes, this property is enough to make sure that computation never gets stuck. Having (mutual) recursive processes and fixed points, however, this property is not strong enough to restrict internal loops. The composition $x \leftarrow$ `Loop` $\leftarrow \cdot \mid y \leftarrow$ `Block` $\leftarrow x$, for example, satisfies the progress property but it never interacts with any other external process. We introduce a stronger form of the progress property, in the sense that it requires any of conditions (ii) or (iii) to hold after a finite number of computation steps.

Like cut elimination, strong progress is not compositional. Processes `Loop` and `Block` both satisfy the strong progress property but their composition $x \leftarrow$ `Loop` $\leftarrow \cdot \mid y \leftarrow$ `Block` $\leftarrow x$ does not. We will show in Section 12 that FS validity implies strong progress. But, in contrast to strong progress, this condition is compositional in the sense that composition of two disjoint valid programs is also valid. FS validity is not local. Our goal is to construct a locally checkable validity condition that accepts (a subset of) programs satisfying strong progress and is compositional.

In functional programming languages, a program is called *terminating* if it reduces to a value in a finite number of steps, and is called *productive* if every piece of the output is generated in finite number of steps (even if the program potentially runs forever). The theoretical underpinnings for terminating and productive programs are also least and greatest fixed points, respectively, but due to the functional nature of computation they take a different and less symmetric form than here (see, for example, [BM13, Gra16]).

In our system of session types, least and greatest fixed points correspond to defined type variables with positive and negative polarity, respectively, and their behaviors are quite symmetric: As in Definition 2, an unfolding message μ_t for a type variable t with positive polarity is received from the left and sent to the right, while for a variable t with negative polarity, the unfolding message ν_t is received from the right and sent to the left. Back to Examples 4 and 5, process `Loop` seems less acceptable than process `Block`: process `Loop` does not receive any least or greatest fixed point unfolding messages and its circularity cannot be justified with either induction or co-induction. We want our algorithm to accept process `Block` rather than `Loop`, since it cannot accept both. This motivates a unified definition of termination and productivity in the session types system based on its symmetry.

Definition 3. A program is *(finitely) reactive to the left* if it receives every fixed point unfolding message μ_t from the *left*, if any, in a finite number of steps, and it stops in finite steps if there are no more fixed point unfolding messages to be received from the *left*.
A program is *(finitely) reactive to the right* if it receives every fixed point unfolding message ν_t from the *right*, if any, in a finite number of steps, and it stops in finite steps if there are no more unfolding messages to be received from the *right*.
A program is called *(finitely) reactive* if it is either *reactive to the right* or *to the left*.

By this definition, process `Block` is reactive while process `Loop` is not. Although reactivity is not compositional we use it to explain the intuition behind our condition and construct it one step at a time. Later, in Sections 11 and 12 we prove that our algorithm ensures FS validity and strong progress.

We conclude this section with an example of a reactive process `Copy`. This process, similar to `Block`, receives a natural number from left but instead of consuming it, sends it over to the right along a channel of type nat.

Example 6. With signature to be

$$\Sigma_1 := \mathsf{nat} =^1_\mu \oplus\{z : \mathbf{1}, s : \mathsf{nat}\}$$

we defined process `Copy`

$$x : \mathsf{nat} \vdash \mathtt{Copy} :: (y : \mathsf{nat})$$

as

```
$y \leftarrow \texttt{Copy} \leftarrow x =$
    case $Lx$ ($\mu_{nat} \Rightarrow$                    % receive $\mu_{nat}$ from left      (i)
        case $Lx$                                    % receive a label from left     (ii)
            ( $z \Rightarrow Ry.\mu_{nat}$;                   % send $\mu_{nat}$ to right         (ii-a)
                    $Ry.z$;                          % send label $z$ to right
                    wait $Lx$;                       % wait and close $x$
                        close $Ry$                   % close $y$
            | $s \Rightarrow Ry.\mu_{nat}$;                   % send $\mu_{nat}$ to right         (ii-b)
                    $Ry.s$;                          % send label $s$ to right
                    $y \leftarrow \texttt{Copy} \leftarrow x$))              % recursive call
```

This is an example of a recursive process, and $\mathcal{P}_3 := \langle\{\texttt{Copy}\}, \texttt{Copy}\rangle$ forms a *reactive to the left* program over the signature Σ_1:
(i) It waits until it receives a *positive* fixed point unfolding message from the left, (ii) waits for another message from the left to determine the path it will continue with:

(a) If the message is a z label, (ii-a) the program sends a *positive* fixed point unfolding message to the right, following by a message of label z to the right, and then waits until a closing message is received from the left. Upon receiving that message, it closes the right channel.

(b) If the message is a s label, (ii-b) the program sends a *positive* fixed point unfolding message to the right, following by a message of label s to the right, and then calls itself and loops back to (i). The computational content of Copy is to simply copy a natural number given from the left to the right. Process Copy is cut-free and satisfies the strong progress property. This property is preserved when composed with Block as $y \leftarrow \texttt{Copy} \leftarrow x \mid z \leftarrow \texttt{Block} \leftarrow y$. △

5. Local Validity Algorithm: Naive Version

In this section we develop a first naive version of our local validity algorithm using Examples 7-8.

Example 7. Let the signature be

$$\Sigma_2 := \mathsf{bits} =^1_\mu \oplus\{b0 : \mathsf{bits},\ b1 : \mathsf{bits}\}$$

and define the process BitNegate

$$x : \mathsf{bits} \vdash \texttt{BitNegate} :: (y : \mathsf{bits})$$

with

```
$y \leftarrow \texttt{BitNegate} \leftarrow x =$
    $\textbf{case}\ Lx\ (\mu_{bits} \Rightarrow$                 % receive $\mu_{bits}$ from left          (i)
        $\textbf{case}\ Lx$                                   % receive a label from left        (ii)
          $(\ b0 \Rightarrow Ry.\mu_{bits};$                  % send $\mu_{bits}$ to right          (ii-a)
                  $Ry.b1;$                                    % send label b1 to right
                   $y \leftarrow \texttt{BitNegate} \leftarrow x$       % recursive call
          $|\ b1 \Rightarrow Ry.\mu_{bits};$                  % send $\mu_{bits}$to right           (ii-b)
                  $Ry.b0;$                                    % send label b0 to right
                   $y \leftarrow \texttt{BitNegate} \leftarrow x))$     % recursive call
```

$\mathcal{P}_4 := \langle\{\texttt{BitNegate}\}, \texttt{BitNegate}\rangle$ forms a *reactive to the left* program over the signature Σ_2:
(i) It waits until it receives a *positive* fixed point unfolding message from the left, (ii) waits for another message from the left to determine the path it shall continue with:

(a) If the message is a *b0* label, (ii-a) the program sends a *positive* fixed point unfolding message to the right, following by a message of label *b1* to the right, and then calls itself recursively and loops back to (i).

(b) If the message is a *b0* label, similarly, (ii-b) the program sends a *positive* fixed point unfolding message to the right, following by a message of label *b1* to the right, and then calls itself and loops back to (i).

Computationally, `BitNegate` is a buffer with one bit capacity that receives a bit from the left and stores it until a process on its right asks for it. After that, the bit is negated and sent to the right and the buffer becomes free to receive another bit. △

Example 8. Dual to Example 7, we can define `coBitNegate`:
Let the signature be

$$\Sigma_3 := \mathsf{cobits} =^1_\nu \&\{b0 : \mathsf{cobits},\ b1 : \mathsf{cobits}\}$$

with process

$$x : \mathsf{cobits} \vdash \texttt{coBitNegate} :: (y : \mathsf{cobits})$$

where `coBitNegate` is defined as

```
$y \leftarrow \texttt{coBitNegate} \leftarrow x =$
    $\textbf{case}\ Ry\ (\nu_{cobits} \Rightarrow$               % receive $\nu_{cobits}$ from right       (i)
        $\textbf{case}\ Ry$                                   % receive a label from right       (ii)
          $(\ b0 \Rightarrow Lx.\nu_{cobits};$                % send $\nu_{cobits}$ to left           (ii-a)
                  $Lx.b1;$                                    % send label b1 to left
                   $y \leftarrow \texttt{coBitNegate} \leftarrow x$     % recursive call
          $|\ b1 \Rightarrow Lx.\nu_{cobits};$                % send $\nu_{cobits}$ to left           (ii-b)
                  $Lx.b0;$                                    % send label b0 to left
                   $y \leftarrow \texttt{coBitNegate} \leftarrow x))$   % recursive call
```

$\mathcal{P}_5 := \langle\{\texttt{coBitNegate}\}, \texttt{coBitNegate}\rangle$ forms a *reactive to the right* program over the signature Σ_3:
(i) It waits until it receives a *negative* fixed point unfolding message from the right, (ii) waits for another message from the right to determine the path it shall continue with:

(a) If the message is a *b0* label, (ii-a) the program sends a *negative* fixed point unfolding message to the left, following by a message of label $b1$ to the left, and then calls itself recursively and loops back to (i).

(b) If the message is a $b1$ label, similarly, (ii-b) the program sends a *negative* fixed point unfolding message to the left, following by a message of label $b0$ to the left, and then calls itself and loops back to (i).

Computationally, `coBitNegate` is a buffer with one bit capacity. In contrast to `BitNegate` in Example 7, its types have negative polarity: it receives a bit from the *right*, and stores it until a process on its *left* asks for it. After that the bit is negated and sent to the *left* and the buffer becomes free to receive another bit. △

Remark 1. The property that assures the reactivity of the previous examples lies in their step (i) in which the program *waits* for an unfolding message from the left/right, i.e., the program can only continue the computation if it *receives* a message at step (i), and even after receiving the message it can only take finitely many steps further before the computation ends or another unfolding message is needed.

We first develop a naive version of our algorithm which captures the property explained in Remark 1: Associate an initial integer value (say 0) to each channel and define the basic step of our algorithm to be *decreasing* the value associated to a channel *by one* whenever it *receives* a fixed point unfolding message. Also, for a reason that is explained later in Remark 2, whenever a channel *sends* a fixed point unfolding message its value is *increased by one*. Then at each recursive call, the value of the left and right channels are compared to their initial value.

For instance, in Example 6, in step (i) where the process receives a μ_{nat} message via the left channel (x), the value associated with x is decreased by one, while in steps (ii-a) and (ii-b) in which the process sends a μ_{nat} message via the right channel (y) the value associated with y is increased by one:

$$\begin{array}{lrr}
 & x & y \\
y \leftarrow \mathtt{Copy} \leftarrow x = & 0 & 0 \\
\qquad\qquad \mathbf{case}\, Lx\,(\mu_{nat} \Rightarrow & -1 & 0 \\
\qquad\qquad\qquad \mathbf{case}\, Lx\,(z \Rightarrow Ry.\mu_{nat}; & -1 & 1 \\
\qquad\qquad\qquad\qquad R.z; \mathbf{wait}\, Lx; \mathbf{close}\, Ry & -1 & 1 \\
\qquad\qquad\qquad\quad s \Rightarrow Ry.\mu_{nat}; & -1 & 1 \\
\qquad\qquad\qquad\qquad Ry.s; y \leftarrow \mathtt{Copy} \leftarrow x)) & -1 & 1
\end{array}$$

When the recursive call occurs, channel x has the value $-1 < 0$, meaning that at some point in the computation it received a positive fixed point unfolding message. We can simply compare the value of the list $[x, y]$ lexicographically at the beginning and just before the recursive call: $[-1, 1]$ being less than $[0, 0]$ exactly captures the property observed in Remark 1 for the particular signature Σ_1. Note that by the definition of Σ_1, y never receives a fixed point unfolding message, so its value never decreases, and x never sends a fixed point unfolding message, and its value never increases.
The same criteria works for the program $\mathcal{P}_3$ over the signature Σ_2 defined in Example 7, since Σ_2

also contains only one positive fixed point:

$$\begin{array}{llrr}
 & & x & y \\
y \leftarrow \texttt{BitNegate} \leftarrow x = & & 0 & 0 \\
\quad \textbf{case}\, Lx\ (\mu_{bits} \Rightarrow & & -1 & 0 \\
\quad\quad \textbf{case}\, Lx\ (b0 \Rightarrow Ry.\mu_{bits}; & & -1 & 1 \\
\quad\quad\quad Ry.b1; y \leftarrow \texttt{BitNegate} \leftarrow x & & -1 & 1 \\
\quad\quad b1 \Rightarrow Ry.\mu_{bits}; & & -1 & 1 \\
\quad\quad\quad Ry.b0; y \leftarrow \texttt{BitNegate} \leftarrow x)) & & -1 & 1
\end{array}$$

At both recursive calls the value of the list $[x, y]$ is less than $[0, 0]$: $[-1, 1] < [0, 0]$.
However, for a program defined on a signature with a negative polarity such as the one defined in Example 8, this condition does not work:

$$\begin{array}{llrr}
 & & x & y \\
y \leftarrow \texttt{coBitNegate} \leftarrow x = & & 0 & 0 \\
\quad \textbf{case}\, Ry\ (\nu_{cobits} \Rightarrow & & 0 & -1 \\
\quad\quad \textbf{case}\, Ry\ (b0 \Rightarrow Lx.\nu_{cobits}; & & 1 & -1 \\
\quad\quad\quad Lx.b1; y \leftarrow \texttt{coBitNegate} \leftarrow x & & 1 & -1 \\
\quad\quad b1 \Rightarrow Lx.\nu_{cobits}; & & 1 & -1 \\
\quad\quad\quad Lx.b0; y \leftarrow \texttt{coBitNegate} \leftarrow x)) & & 1 & -1
\end{array}$$

By the definition of Σ_3, y only receives unfolding fixed point messages, so its value only decreases. On the other hand, x cannot receive an unfolding fixed point from the left and thus its value never decreases. In this case the property in Remark 1 is captured by comparing the initial value of the list $[y, x]$, instead of $[x, y]$, with its value just before the recursive call: $[-1, 1] < [0, 0]$.

For a signature with only a single recursive type we can form a list by looking at the polarity of its type such that the value of the channel that receives the unfolding message comes first, and the value of the other one comes second.

6. Priorities in the Local Validity Algorithm

The property given in Remark 1 of previous section is not strict enough, particularly when the signature has more than one recursive type. In that case not all programs that are waiting for a fixed point unfolding message before a recursive call are reactive.

Example 9. Consider the signature

$$\begin{aligned}
\Sigma_4 := \mathsf{ack} &=^1_\mu \oplus\{ack : \mathsf{astream}\}, \\
\mathsf{astream} &=^2_\nu \&\{head : \mathsf{ack},\ \ tail : \mathsf{astream}\}, \\
\mathsf{nat} &=^3_\mu \oplus\{z : 1,\ \ s : nat\}
\end{aligned}$$

ack is a type with *positive* polarity that, upon unfolding, sends an *acknowledgment* message to the right (or receives it from the left). $\mathsf{astream}$ is a type with *negative* polarity of a potentially infinite stream where its *head* is always followed by an acknowledgement.

$\mathcal{P}_6 := \langle\{\texttt{Ping}, \texttt{Pong}, \texttt{PingPong}\}, \texttt{PingPong}\rangle$ forms a program over the signature Σ_4 with the typing of its processes

$$x : \mathsf{nat} \vdash \texttt{Ping} :: (w : \mathsf{astream})$$
$$w : \mathsf{astream} \vdash \texttt{Pong} :: (y : \mathsf{nat})$$
$$x : \mathsf{nat} \vdash \texttt{PingPong} :: (y : \mathsf{nat})$$

We define processes Ping, Pong, and PingPong over Σ_4 as:

```
$y \leftarrow \texttt{PingPong} \leftarrow x =$
    $w \leftarrow \texttt{Ping} \leftarrow x;$                    % spawn process Ping                (i)
        $y \leftarrow \texttt{Pong} \leftarrow w$                % continue with a tail call

$y \leftarrow \texttt{Pong} \leftarrow w =$
    $Lw.\nu_{astream};$                                         % send $\nu_{astream}$ to left      (ii-Pong)
      $Lw.head;$                                                % send label head to right          (iii-Pong)
        case $Lw$ $(\mu_{ack} \Rightarrow$                      % receive $\mu_{ack}$ from left     (iv-Pong)
           case $Lw$ (                                          % receive a label from left
                    $ack \Rightarrow Ry.\mu_{nat};$             % send $\mu_{nat}$ to right
                       $Ry.s;$                                  % send label s to right
                          $y \leftarrow \texttt{Pong} \leftarrow w))$   % recursive call

$w \leftarrow \texttt{Ping} \leftarrow x =$
    case $Rw$ $(\nu_{astream} \Rightarrow$                      % receive $\nu_{astream}$ from right   (ii-Ping)
       case $Rw$ (                                              % receive a label from right
               $head \Rightarrow Rw.\mu_{ack};$                 % send $\mu_{ack}$ to right         (iii-Ping)
                         $Rw.ack;$                              % send label ack to right
                            $w \leftarrow \texttt{Ping} \leftarrow x$   % recursive call
               $\mid tail \Rightarrow w \leftarrow \texttt{Ping} \leftarrow x))$   % recursive call
```

(i) Program $\mathcal{P}_6$ starting from PingPong, spawns a new process Ping and continues as Pong:

(ii-Pong) Process Pong sends an astream unfolding and then a *head* message to the left, and then (iii-Pong) *waits* for an acknowledgment, i.e., *ack*, from the left.

(ii-Ping) At the same time process Ping *waits* for an astream fixed point unfolding message from the right, which becomes available after step (ii-Pong). Upon receiving the message, it waits for receiving either *head* or *tail* from the right, which is also available from (ii-Pong) and is actually a *head*. So (iii-Ping) it continues with the path corresponding to *head*, and acknowledges receipt of the previous messages by sending an unfolding messages and the label *ack* to the right, and then it calls itself (ii-Ping).

(iv-Pong) Process Pong now receives the two messages sent at (iii-Ping) and thus can continue by sending a nat unfolding message and the label *s* to the right, and finally calling itself (ii-Pong). Although both recursive processes Ping and Pong at some point *wait* for a fixed point unfolding message, this program runs infinitely without receiving any messages from the outside, and thus is not reactive. △

The back-and-forth exchange of fixed point unfolding messages between two processes in the previous example can arise when at least two mutually recursive types with different polarities are in the signature. This is why we need to incorporate priorities of the type variables into the validity checking algorithm.

Remark 2. In Example 9, for instance, we can add the condition that the *wait* in step (ii-Ping) on *receiving* an unfolding message $\nu_{astream}$ for a type variable with priority 2 is not valid anymore after step (iii-Ping), since the process *sends* an unfolding message μ_{ack} for a type variable with a higher priority (priority 1) at step (iii-Ping).

To include such a condition in our algorithm we form a list for each process. This list stores the information of the fixed point unfolding messages that the process received and sent before a recursive call for each type variable in their order of priority.

Example 10. Consider the signature and program $\mathcal{P}_6$ as defined in Example 9. For the process $x : \mathsf{nat} \vdash w \leftarrow \texttt{Ping} \leftarrow x :: (w : \mathsf{astream})$ form the list

$$[\mathsf{ack} - \textit{received}, \mathsf{ack} - \textit{sent}, \mathsf{astream} - \textit{received}, \mathsf{astream} - \textit{sent}, \mathsf{nat} - \textit{received}, \mathsf{nat} - \textit{sent}].$$

Types with positive polarity, i.e., ack and nat, receive messages from the left channel (x) and send messages to the right channel (w), while those with negative polarity, i.e., astream, receive from the right channel (w) and send to the left one (x). Thus, the above list can be rewritten as

$$[x_{\mathsf{ack}}, w_{\mathsf{ack}}, w_{\mathsf{astream}}, x_{\mathsf{astream}}, x_{\mathsf{nat}}, w_{\mathsf{nat}}].$$

To keep track of the sent/received messages, we start with $[0, 0, 0, 0, 0, 0]$ as the value of the list, when the process $x : \mathsf{nat} \vdash \texttt{Ping} :: (w : \mathsf{astream})$ is first spawned. Then, similar to what we had in the naive version of the algorithm, on the steps in which the process *receives* a fixed point unfolding message, the value of the corresponding element of the list is *decreased by one*. And on the steps it *sends* a fixed point unfolding message, the corresponding value is *increased by one*:

```
w ← Ping ← x =                                              [0, 0 , 0 , 0, 0, 0]
      case Rw (ν_astream ⇒                                  [0, 0, −1, 0, 0, 0]
            case Rw (head ⇒ Rw.μ_ack;                       [0, 1, −1, 0, 0, 0]
                         Rw.ack; w ← Ping ← x               [0, 1, −1, 0, 0, 0]
                   | tail ⇒ w ← Ping ← x))                  [0, 0, −1, 0, 0, 0]
```

The two last lines are the values of the list on which process Ping calls itself recursively. The validity condition as described in Remark 2 holds iff the value of the list at the time of the recursive call is less than the value the process started with, in the lexicographical order. Here, for example, $[0, 1, -1, 0] \nless [0, 0, 0, 0]$, and the validity condition does not hold for this recursive call. △

The following definition captures the idea of forming lists described above.

Definition 4. For a process

$$\bar{x} : \omega \vdash P :: (y : B),$$

over the signature Σ, define $list(\bar{x}, y) = [v_i]_{\,i \leq n}$ such that

(1) $v_i = (\bar{x}_i, y_i)$ if $\epsilon(i) = \mu$, and
(2) $v_i = (y_i, \bar{x}_i)$ if $\epsilon(i) = \nu$,

where n is the maximum priority in Σ.

In the remainder of this section we use n to denote the maximum priority in Σ.

Example 11. Consider the signature Σ_1 and program $\mathcal{P}_3 := \langle\{\mathsf{Copy}\}, \mathsf{Copy}\rangle$, in Example 6: $\Sigma_1 := \mathsf{nat} =^1_\mu \oplus\{z : 1, s : \mathsf{nat}\}$, and

$$\begin{aligned} y \leftarrow \mathsf{Copy} \leftarrow x = \mathbf{case}\, Lx\, (\mu_{nat} \Rightarrow \mathbf{case}\, Lx\, (\, & z \Rightarrow Ry.\mu_{nat}; Ry.z; \mathbf{wait}\, Lx; \mathbf{close}\, Ry \\ & \mid s \Rightarrow Ry.\mu_{nat}; Ry.s; y \leftarrow \mathsf{Copy} \leftarrow x)) \end{aligned}$$

By Definition 4, for process $x : \mathsf{nat} \vdash \mathsf{Copy} :: (y : \mathsf{nat})$, we have $n = 1$, and $list(x, y) = [(x_1, y_1)]$ since $\epsilon(1) = \mu$. Just as for the naive version of the algorithm, we can trace the value of $list(x, y)$:

$$\begin{array}{lr} y \leftarrow \mathsf{Copy} \leftarrow x = & [0, 0] \\ \qquad \mathbf{case}\, Lx\, (\mu_{nat} \Rightarrow & [-1, 0] \\ \qquad\qquad \mathbf{case}\, Lx\, (z \Rightarrow Ry.\mu_{nat}; & [-1, 1] \\ \qquad\qquad\qquad R.z; \mathbf{wait}\, Lx; \mathbf{close}\, Ry & [-1, 1] \\ \qquad\qquad \mid s \Rightarrow Ry.\mu_{nat}; & [-1, 1] \\ \qquad\qquad\qquad Ry.s; y \leftarrow \mathsf{Copy} \leftarrow x & [-1, 1] \end{array}$$

Here, $[-1, 1] < [0, 0]$ and the recursive call is classified as valid. △

Sometimes we are interested in a prefix of the list from Definition 4. We give the following definition of $list(x, y, j)$ to crop $list(x, y)$ exactly before the element corresponding to a *sent* fixed point unfolding message of types with priority j. These prefixes are used later in Definition 7.

Definition 5. For a process

$$\bar{x} : \omega \vdash P :: (y : B)$$

over signature Σ, and $0 \leq j \leq n$, define $list(\bar{x}, y, j)$, as a prefix of the list $list(\bar{x}, y) = [v_i]_{i \leq n}$ by

(1) $[]$ if $i = 0$,
(2) $[[v_i]_{i<j},\ (\bar{x}_j)]$ if $\epsilon(j) = \mu$,
(3) $[[v_i]_{i<j},\ (y_j)]$ if $\epsilon(j) = \nu$.

Example 12. Consider the signature introduced in Example 9

$$\begin{aligned} \Sigma_4 := \mathsf{ack} &=^1_\mu \oplus\{ack : \mathsf{astream}\}, \\ \mathsf{astream} &=^2_\nu \&\{head : \mathsf{ack},\ \ tail : \mathsf{astream}\}, \\ \mathsf{nat} &=^3_\mu \oplus\{z : 1,\ \ s : nat\} \end{aligned}$$

and program $\mathcal{P}_6 := \langle\{\mathtt{Ping}, \mathtt{Pong}, \mathtt{PingPong}\}, \mathtt{PingPong}\rangle$.
For process $x : \mathsf{nat} \vdash \mathtt{Ping} :: (w : \mathsf{astream})$:

$$\begin{aligned} & list(x, w) = [(x_1, w_1), (w_2, x_2), (x_3, w_3)], \\ & list(x, w, 3) = [(x_1, w_1), (w_2, x_2), (x_3)], \\ & list(x, w, 2) = [(x_1, w_1), (w_2)], \\ & list(x, w, 1) = [(x_1)],\ \ and \\ & list(x, w, 0) = []. \end{aligned}$$

△

To capture the idea of *decreasing/increasing* the value of the elements on $list(\_, \_)$ by *one*, as depicted in Example 10 and Example 11, we assume that a channel transforms to a new generation of itself after sending or receiving a fixed point unfolding message.

Example 13. Process $x : \mathsf{nat} \vdash y \leftarrow \mathsf{Copy} \leftarrow x :: (y : \mathsf{nat})$ in Example 11 starts its computation with the initial generation of its left and right channels:

$$x^0 : \mathsf{nat} \vdash y^0 \leftarrow \mathsf{Copy} \leftarrow x^0 :: (y^0 : \mathsf{nat}).$$

The channels evolve as the process sends or receives a fixed point unfolding message along them:

$$\begin{array}{ll}
y^0 \leftarrow \mathsf{Copy} \leftarrow x^0 = & \\
\quad \mathbf{case}\, Lx^0\ (\mu_{nat} \Rightarrow & x^0 \rightsquigarrow x^1 \\
\qquad \mathbf{case}\, Lx^1\ (z \Rightarrow Ry^0.\mu_{nat}; & y^0 \rightsquigarrow y^1 \\
\qquad\qquad Ry^1.z; \mathbf{wait}\, Ly^1; \mathbf{close}\, Rx^1 & \\
\qquad\quad \mid s \Rightarrow Ry^0.\mu_{nat}; & y^0 \rightsquigarrow y^1 \\
\qquad\qquad Ry^1.s; y^1 \leftarrow \mathsf{Copy} \leftarrow x^1)) &
\end{array}$$

On the last line the process

$$x^1 : \mathsf{nat} \vdash y^1 \leftarrow \mathsf{Copy} \leftarrow x^1 :: (y^1 : \mathsf{nat})$$

is called recursively with a new generation of variables. △

In the inference rules introduced in Section 9, instead of recording the absolute value of each element of the *list*(_, _) as we did in Example 10 and Example 11, we introduce an extra set Ω that stores the relation between different generations of a channel indexed by their priority of types.

Remark 3. Generally speaking, $x_i^{\alpha+1} < x_i^\alpha$ is added to Ω, when x^α *receives* a fixed point unfolding message of a type with priority i and transforms to $x^{\alpha+1}$. This corresponds to the *decrease by one* in the previous examples.

If x^α *sends* a fixed point unfolding message of a type with priority i and evolves to $x^{\alpha+1}$, x_i^α and $x_i^{\alpha+1}$ are considered to be incomparable in Ω. This corresponds to *increase by one* in the previous examples, since for the sake of comparing *list*(_, _) at the *first call* of a process and just before a *recursive call* in a lexicographic order, there is no difference whether $x^{\alpha+1}$ is greater than x^α or incomparable to it.

When x^α receives/sends a fixed point unfolding message of a type with priority i and transforms to $x^{\alpha+1}$, for any type with priority $j \neq i$, the value of x_j^α and $x_j^{\alpha+1}$ must remain equal. In these steps, we add $x_j^\alpha = x_j^{\alpha+1}$ for $j \neq i$ to Ω.

A process in the formalization of the intuition above is therefore typed as

$$x^\alpha : A \vdash_\Omega P :: (y^\beta : B),$$

where x^α is the α-th generation of channel x. The relation between the channels indexed by their priority of types is built step by step in Ω and represented by ≤. The reflexive transitive closure of Ω forms a partial order $\leq_\Omega$. We extend $\leq_\Omega$ to the *list* of channels indexed by the priority of their types considered lexicographically. We may omit subscript Ω from $\leq_\Omega$ whenever it is clear from the context.

7. Mutual Recursion in the Local Validity Condition

In examples of previous sections, the recursive calls were not *mutual*. In the general case, a process may call any other process variable in the program, and this call can be mutually recursive. In this section, we incorporate mutual recursive calls into our algorithm.

Example 14. Recall signature Σ_4 from Example 9

$$\begin{aligned}\Sigma_4 := \mathsf{ack} &=^1_\mu \oplus\{ack : \mathsf{astream}\},\\ \mathsf{astream} &=^2_\nu \&\{head : \mathsf{ack},\ \ tail : \mathsf{astream}\},\\ \mathsf{nat} &=^3_\mu \oplus\{z : 1,\ \ s : nat\}\end{aligned}$$

Define program $\mathcal{P}_7 = \langle\{\mathtt{Idle}, \mathtt{Producer}, \}, \mathtt{Producer}\rangle$, where

$$\begin{array}{l} z : \mathsf{ack} \vdash w \leftarrow \mathtt{Idle} \leftarrow z :: (w : \mathsf{nat})\\ x : \mathsf{astream} \vdash y \leftarrow \mathtt{Producer} \leftarrow x :: (y : \mathsf{nat}),\end{array}$$

and processes `Idle` (`I`) and `Producer` (or simply `P`) are defined as:

$$w \leftarrow \mathtt{I} \leftarrow z = \mathbf{case}\, Lz^{\alpha}\ (\mu_{ack} \Rightarrow \mathbf{case}\, Lz^{\alpha+1}\ (ack \Rightarrow Rw^{\beta}.\mu_{nat}; Rw^{\beta+1}.s; y^{\beta+1} \leftarrow \mathtt{P} \leftarrow x^{\alpha+1}))$$
$$y \leftarrow \mathtt{P} \leftarrow x = Lx^{\alpha}.\nu_{astream}; Lx^{\alpha+1}.head; y^{\beta} \leftarrow \mathtt{I} \leftarrow x^{\alpha+1}.$$

We calculusate $list(x, y) = [(x_1, y_1), (y_2, x_2), (x_3, y_3)]$ and $list(z, w) = [(z_1, w_1), (w_2, z_2), (z_3, w_3)]$ since $\epsilon(1) = \epsilon(3) = \mu$ and $\epsilon(2) = \nu$.

By analyzing the behavior of this program step by step, we see that it is a *reactive* program that counts the number of acknowledgements received from the left. The program starts with the process $x^0 : \mathsf{astream} \vdash_{\emptyset} y^0 \leftarrow \mathtt{Producer} \leftarrow x^0 :: (y^0 : \mathsf{nat})$. It first sends one message to left to unfold the *negative* fixed point type, and its left channel evolves to a next generation. Then another message is sent to the left to request the *head* of the stream and after that it calls process $y^0 \leftarrow \mathtt{Idle} \leftarrow x^1$.

$$\begin{array}{lll} y^0 \leftarrow \mathtt{Producer} \leftarrow x^0 = & [0,0,0,0,0,0] & \\ \quad Lx^0.\nu_{astream}; & [0,0,0,1,0,0] & x^1_1 = x^0_1, x^1_3 = x^0_3\\ \quad\quad Lx^1.head; y^0 \leftarrow \mathtt{Idle} \leftarrow x^1 & [0,0,0,1,0,0] & \end{array}$$

Process $x^1 : \mathsf{ack} \vdash y^0 \leftarrow \mathtt{Idle} \leftarrow x^1 :: (y^0 : \mathsf{nat})$, then *waits* to receive an acknowledgment from the left via a *positive* fixed point unfolding message for ack and its left channel transforms to a new generation upon receiving it. Then it waits for the label *ack*, and upon receiving it, it sends one message to the right to unfold the *positive* fixed point nat (and this time the right channel evolves). Then it sends the label s to the right and calls $y^1 \leftarrow \mathtt{Producer} \leftarrow x^2$ recursively:

$$\begin{array}{lll} y^0 \leftarrow \mathtt{Idle} \leftarrow x^1 = & [0,0,\ 0,\ 1,\ 0,0] & \\ \quad \mathbf{case}\, Lx^1\ (\mu_{ack} \Rightarrow & [-1,0,0,1,0,0] & x^2_1 < x^1_1, x^2_2 = x^1_2, x^2_3 = x^1_3\\ \quad\quad \mathbf{case}\, Lx^2\ (ack \Rightarrow Ry^0.\mu_{nat}; & [-1,0,0,1,0,1] & y^1_1 = y^0_1, y^1_2 = y^0_2\\ \quad\quad\quad Ry^1.s; y^1 \leftarrow \mathtt{Producer} \leftarrow x^2)) & [-1,0,0,1,0,1] & \end{array}$$

Observe that the actual recursive call for `Producer` occurs at the red line above, where `Producer` eventually calls itself. At that point the absolute value of $list(x^2, y^1)$ is recorded as $[-1, 0, 0, 1, 0, 1]$, which is less than the absolute value of $list(x^0, y^0)$ when `Producer` was called for the first time:

$$[-1, 0, 0, 1, 0, 1] < [0, 0, 0, 0, 0, 0].$$

The same observation can be made by considering the relations introduced in the last column

$$list(x^2, y^1) = [(x^2_1, y^1_1), (y^1_2, x^2_2), (x^2_3, y^1_3)] < [(x^0_1, y^0_1), (y^0_2, x^0_2), (x^0_3, y^0_3)] = list(x^0, y^0)$$

since $x_1^2 < x_1^1 = x_1^0$. This recursive call is valid regardless of the fact that $[0,0,0,1,0,0] \not< [0,0,0,0,0,0]$, i.e.

$$list(x^1, y^0) = [(x_1^1, y_1^0), (y_2^0, x_2^1), (x_3^1, y_3^0)] \not< [(x_1^0, y_1^0), (y_2^0, x_2^0), (x_3^0, y_3^0)] = list(x^0, y^0)$$

since $x_1^1 = x_1^0$ but x_2^1 is incomparable to x_2^0. Similarly, we can observe that the actual recursive call on `Idle`, where `Idle` eventually calls itself, is valid.

To account for this situation, we introduce an order on *process variables* and trace the last seen variable on the path leading to the recursive call. In this example we define `Idle` to be less than `Producer` at position 2 ($\mathtt{I} \subset_2 \mathtt{P}$), i.e.:

> We incorporate process variables `Producer` and `Idle` to the lexicographical order of $list(\_,\_)$ such that their values are placed exactly before the element in the list corresponding to the *sent* unfolding messages of the type with priority 2.

We now trace the ordering as follows:

$y^0 \leftarrow \mathtt{Producer} \leftarrow x^0 =$	$[0,0,0,\mathtt{P},0,0,0]$	
$\quad Lx^0.\nu_{astream};$	$[0,0,0,\mathtt{P},1,0,0]$	$x_1^1 = x_1^0, x_3^1 = x_3^0$
$\quad\quad Lx^1.head; y^0 \leftarrow \mathtt{Idle} \leftarrow x^1$	$[0,0,0,\mathtt{I},1,0,0]$	

$y^0 \leftarrow \mathtt{Idle} \leftarrow x^1 =$	$[0,0,0,\mathtt{I},1,0,0]$	
$\quad$ **case** $Lx^1(\mu_{ack} \Rightarrow$	$[-1,0,0,\mathtt{I},1,0,0]$	$x_1^2 < x_1^1, x_2^2 = x_2^1, x_3^2 = x_3^1$
$\quad\quad$ **case** $Lx^2(ack \Rightarrow Ry^0.\mu_{nat};$	$[-1,0,0,\mathtt{I},1,0,1]$	$y_1^1 = y_1^0, y_2^1 = y_2^0$
$\quad\quad\quad Ry^1.s; y^1 \leftarrow \mathtt{Producer} \leftarrow x^2$	$[-1,0,0,\mathtt{P},1,0,1]$	

$[-1,0,0,\mathtt{P},1,0,1] < [0,0,0,\mathtt{I},1,0,0]$ and $[0,0,0,\mathtt{I},1,0,0] < [0,0,0,\mathtt{P},0,0,0]$ hold, and both mutually recursive calls are recognized to be valid, as they are, without a need to search for actual recursive calls, i.e., where a process calls itself. △

However, not every relation over the process variables forms a partial order. For instance, having both $\mathtt{P} \subset_2 \mathtt{I}$ and $\mathtt{I} \subset_2 \mathtt{P}$ violates the antisymmetry condition. Introducing the position of process variables into the $list(\_,\_)$ is also a delicate issue. For example, if we have both $\mathtt{I} \subset_1 \mathtt{P}$ and $\mathtt{I} \subset_2 \mathtt{P}$, it is not determined where to insert the value of `Producer` and `Idle` on the $list(\_,\_)$. Definition 6 captures the idea of Example 14, while ensuring that the introduced relation is always a well-defined order and it is determined on which position of $list(\_,\_)$ the process variables shall be inserted. Definition 7 gives the lexicographic order on $list(\_,\_)$ augmented with the $\subseteq$ relation.

Definition 6. For the signature Σ and each $0 \leq i \leq n$, let $\subseteq_i$ be a relation on process variables in V that satisfies the following conditions, where (a) $X =_i Y$ iff $X \subseteq_i Y$ and $Y \subseteq_i X$, and (b) $X \subset_i Y$ iff $X \subseteq_i Y$ but $X \neq_i Y$:

(1) $(\forall i \leq n)\,(\forall F \in V)\; F =_i F$,
(2) $(\forall i \leq n)\,(\forall F, G \in V)$ if $F \subseteq_i G$ and $G \subseteq_i F$, then $F =_i G$,
(3) $(\forall i \leq n)\,(\forall F, G \in V)$ if $F \subseteq_i G$ and $G \subseteq_i H$, then $F \subseteq_i H$,
(4) $(\forall i, j \leq n)\,(\forall F, G, H \in V)$ if $F \subseteq_i G$ and $G \subseteq_j H$, then $i = j$,
(5) $(\forall i, j \leq n)\,(\forall F, G \in V)$ if $F \subset_i G$ and $F \subset_j G$ then $j = i$.
(6) $(\forall i \leq n)\,(\forall F, G \in V)$ if $F =_i G$, then $(\forall j \leq n)\; F =_j G$,
(7) $\forall F, G \in V$, either $\exists i \leq n, F \subseteq_i G$ or $\exists i \leq n, G \subseteq_i F$.

Conditions 1-3, guarantee that $\subseteq_i$ is a partial order, and thus $\subset_i$ is the strict partial order defined upon $\subseteq_i$.

Define $\subseteq$ to be $\bigcup_{i \leq n} \subseteq_i$, i.e. $F \subseteq G$ iff $F \subseteq_i G$ for some $i \leq n$. Observe that $\subseteq$ is a total (condition 7) order: it is reflexive (conditions 4 and 1), antisymmetric (conditions 4 and 2), and transitive (conditions 4 and 3). And $\subset$ is the strict portial order derived from it.

Definition 7. Using $\subset$ and $\leq$, we define a new combined order $(\subset, <)$ (used in the local validity condition in Section 10).

$$F, list(\bar{x}, y)\ (\subset, <)\ G, list(\bar{z}, w)$$

iff

(1) If $F \subset G$, i.e., $F \subset_i G$ for a unique i (by condition 5 in Definition 6), then $list(\bar{x}, y, i) \leq list(\bar{z}, w, i)$, otherwise,
(2) if $G \subset F$, i.e. $G \subset_i F$ for a unique i (by condition 5 in Definition 6), then $list(\bar{x}, y, i) < list(\bar{z}, w, i)$, otherwise
(3) $list(\bar{x}, y) < list(\bar{z}, w)$.

By conditions 5 and 6 in Definition 6, $(\subset, <)$ is an irreflexive and transitive relation and thus a strict partial order.

8. A Modified Rule for Cut

There is a subtle aspect of local validity that we have not discussed yet. We need to relate a fresh channel, created by spawning a new process, with the previously existing channels. Process $(x \leftarrow P_x; Q_x)$, for example, creates a fresh channel w^0, spawns process $[w^0/x]P_x$ providing along channel w^0, and then continues as $[w^0/x]Q_x$. For the sake of our algorithm, we need to identify the relation between w^0, x, and y. Since w^0 is a fresh channel, a naive idea is to make w^0 incomparable to any other channel for any type variable $t \in \Sigma$.[1] While sound, we will see in Example 15 that we can improve on this naive approach to cover more valid processes.

Example 15. Define the signature

$$\Sigma_5 := \mathsf{ctr} =^1_\nu \&\{inc : \mathsf{ctr},\ \ val : \mathsf{bin}\},$$
$$\mathsf{bin} =^2_\mu \oplus\{b0 : \mathsf{bin}, b1 : \mathsf{bin}, \$: 1\}.$$

which provides numbers in binary representation as well as an interface to a counter. We explore the following program $\mathcal{P}_8 = \langle\{\texttt{BinSucc}, \texttt{Counter}, \texttt{NumBits}, \texttt{BitCount}\}, \texttt{BitCount}\rangle$, where

$$x : \mathsf{bin} \vdash y \leftarrow \texttt{BinSucc} \leftarrow x :: (y : \mathsf{bin})$$
$$x : \mathsf{bin} \vdash y \leftarrow \texttt{Counter} \leftarrow x :: (y : \mathsf{ctr})$$
$$x : \mathsf{bin} \vdash y \leftarrow \texttt{NumBits} \leftarrow x :: (y : \mathsf{bin})$$
$$x : \mathsf{bin} \vdash y \leftarrow \texttt{BitCount} \leftarrow x :: (y : \mathsf{ctr})$$

We define the relation $\subset$ on process variables as $\texttt{BinSucc} \subset_0 \texttt{Counter} \subset_0 \texttt{NumBits} \subset_0 \texttt{BitCount}$. The process definitions are as follows, shown here already with their termination analysis.

[1]To represent this incomparability in our examples we write "∞" for the value of the fresh channel.

```
$w^\beta \leftarrow \texttt{BinSucc} \leftarrow z^\alpha =$                                    $[0, 0, 0, 0]$
    $\textbf{case}\, Lz^\alpha\ (\mu_{bin} \Rightarrow$                                            $[0, 0, -1, 0]$  $z_1^{\alpha+1} = z_1^\alpha, z_2^{\alpha+1} < z_2^\alpha$
        $\textbf{case}\, Lz^{\alpha+1}\ (b0 \Rightarrow Rw^\beta.\mu_{bin};$                        $[0, 0, -1, 1]$  $w_1^{\beta+1} = w_1^\beta$
                $Rw^{\beta+1}.b1; w^{\beta+1} \leftarrow z^{\alpha+1}$                              $[0, 0, -1, 1]$
            $\mid b1 \Rightarrow Rw^\beta.\mu_{bin};$                                           $[0, 0, -1, 1]$  $w_1^{\beta+1} = w_1^\beta$
                $Rw^{\beta+1}.b0; w^{\beta+1} \leftarrow \texttt{BinSucc} \leftarrow z^{\alpha+1}$                 $[0, 0, -1, 1]$
            $\mid \$ \Rightarrow Rw^\beta.\mu_{\mathsf{bin}}; Rw^{\beta+1}.b1;$                                $[0, 0, -1, 1]$  $w_1^{\beta+1} = w_1^\beta$
                $Rw^{\beta+1}.\mu_{\mathsf{bin}}; Rw^{\beta+2}.\$;\ w^{\beta+2} \leftarrow z^{\alpha+1}))$                $[0, 0, -2, 2]$  $w_1^{\beta+2} = w_1^{\beta+1}$

$y^\beta \leftarrow \texttt{Counter} \leftarrow w^\alpha =$                                    $[0, 0, 0, 0]$
    $\textbf{case}\, Ry^\beta\ (\nu_{ctr} \Rightarrow$                                             $[-1, 0, 0, 0]$  $y_1^{\beta+1} < y_1^\beta, y_2^{\beta+1} = y_2^\beta$
        $\textbf{case}\, Ry^{\beta+1}\ (inc \Rightarrow z^0 \leftarrow \texttt{BinSucc} \leftarrow w^\alpha;$                 $\texttt{BinSucc} \subset_0 \texttt{Counter}$
                $y^{\beta+1} \leftarrow \texttt{Counter} \leftarrow z^0$                              $[-1, \infty, \infty, 0]$
            $\mid val \Rightarrow y^{\beta+1} \leftarrow w^\alpha))$                                    $[-1, 0, 0, 0]$

$w^\beta \leftarrow \texttt{NumBits} \leftarrow x^\alpha =$                                    $[0, 0, 0, 0]$
    $\textbf{case}\, Lx^\alpha\ (\mu_{bin} \Rightarrow$                                            $[0, 0, -1, 0]$  $x_1^{\alpha+1} = x_1^\alpha, x_2^{\alpha+1} < x_2^\alpha$
        $\textbf{case}\, Lx^{\alpha+1}\ (b0 \Rightarrow z^0 \leftarrow \texttt{NumBits} \leftarrow x^{\alpha+1};$           $[?, 0, -1, ?]$  $z_1^0 \overset{?}{=} w_1^\beta, z_2^0 \overset{?}{=} w_2^\beta$
                $w^\beta \leftarrow \texttt{BinSucc} \leftarrow z^0$                                   $\texttt{BinSucc} \subset_0 \texttt{NumBits}$
            $\mid b1 \Rightarrow z^0 \leftarrow \texttt{NumBits} \leftarrow x^{\alpha+1};$                       $[?, 0, -1, ?]$  $z_1^0 \overset{?}{=} w_1^\beta, z_2^0 \overset{?}{=} w_2^\beta$
                  $w^\beta \leftarrow \texttt{BinSucc} \leftarrow z^0$                                 $\texttt{BinSucc} \subset_0 \texttt{NumBits}$
            $\mid \$ \Rightarrow Rw^\beta.\mu_{\mathsf{bin}}; Rw^{\beta+1}.\$;\ w^{\beta+1} \leftarrow x^{\alpha+1}))$               $[0, 0, -1, 1]$

$y^\beta \leftarrow \texttt{BitCount} \leftarrow x^\alpha = w^0 \leftarrow \texttt{NumBits} \leftarrow x^\alpha; y^\beta \leftarrow \texttt{Counter} \leftarrow w^0$
```

The program starts with process `BitCount` which creates a fresh channel w^0, spawns a new process $w^0 \leftarrow \texttt{NumBits} \leftarrow x^\alpha$, and continues as $y^\beta \leftarrow \texttt{Counter} \leftarrow w^0$.

Process $y^\beta \leftarrow \texttt{Counter} \leftarrow w^\alpha$ as its name suggests works as a counter. It waits to receive a fixed point unfolding message from the right. When it receives the unfolding message, if the next message from the right is *val*, the process forwards the binary number offered by w^α to $y^{\beta+1}$. If the next message from the right is *inc*, it spawns a new process `BinSucc` to compute the successor of the binary number offered by channel w^α. Then the incremented binary number offered by z^0 is passed to its continuation $y^{\beta+1} \leftarrow \texttt{Counter} \leftarrow z^0$. Note that in this process, both (mutually) recursive calls pass the algorithm even with the naive approach.

The naive approach also accepts process $w^\beta \leftarrow \texttt{BinSucc} \leftarrow z^\alpha$. This process first receives a fixed point unfolding message from the left. If the next message from the left is bit *b0*, it sends bit *b1* to the right and then forwards the rest of bits offered by $z^{\alpha+1}$ to $w^{\beta+1}$. If the next message is \$, signaling the end of the binary number offered by $z^{\alpha+1}$, it sends bit *b1* to the right and ends the binary message provided along $w^{\beta+1}$. The only recursive call in this process happens when the next message from the left is bit *b1*. In this case, the process sends bit *b0* to the right and calls $w^{\beta+1} \leftarrow \mathtt{BinSucc} \leftarrow z^{\alpha+1}$ recursively to compute successor of the rest of the binary number offered along $z^{\alpha+1}$.

The process $w^{\beta} \leftarrow \mathtt{NumBits} \leftarrow x^{\alpha}$ works on a binary number provided along channel x^{α}. It offers the number of bits of the binary along channel w^{β} as another binary number: `NumBits` first receives a μ unfolding message from the left, then if the process receives an "end of number" message ($), it sends the same message to the right. If it receives a bit (either *b0* or *b1*), then it spawns a new process $z^0 \leftarrow \mathtt{NumBits} \leftarrow x^{\alpha+1}$ to count the remaining number of bits in the binary number. This number is then offered along the fresh channel z^0 to the continuation of the original process as $w^{\beta} \leftarrow \mathtt{BinSucc} \leftarrow z^0$.

Process `NumBits` is a reactive one: It will receive every unfolding message from the left in finite steps and if there is no remaining unfolding messages to receive, it stops. However with our naive approach toward spawning a new process, the recursive calls have the value $[\infty, 0, -1, \infty] \not< [0, 0, 0, 0]$, meaning that they do not satisfy the validity condition.

Note that we cannot just define $z_1^0 = w_1^{\beta}$ and $z_2^0 = w_2^{\beta}$, or $z_1^0 = z_2^0 = 0$. The value of z^0 depends on how this channel evolves in the process $w^{\beta} \leftarrow \mathtt{BinSucc} \leftarrow z^0$. But by definition of type bin, no matter how $z^0 : \mathsf{bin}$ evolves to some z^{η} in process `BinSucc`, it won't be the case that $z^{\eta} : \mathsf{ctr}$. In other words, ctr is not visible from bin and z^0 never evolves to z^{η} such that z_1^{η} has a different value than z_1^0. So in this recursive call, the value of z_1^{η} is not important anymore and we safely put $z_1^0 = w_1^{\beta}$. In this improved version we have:

```
$w^{\beta} \leftarrow \mathtt{NumBits} \leftarrow x^{\alpha} =$                                      $[0,0,\ 0,\ 0]$
  $\mathbf{case}\ Lx^{\alpha}\ (\mu_{bin} \Rightarrow$                                     $[0,0,-1,0]$  $x_1^{\alpha+1} = x_1^{\alpha}, x_2^{\alpha+1} < x_2^{\alpha}$
    $\mathbf{case}\ Lx^{\alpha+1}\ (b0 \Rightarrow z^0 \leftarrow \mathtt{NumBits} \leftarrow x^{\alpha+1};$   $[0,0,-1,\infty]$   $z_1^0 = w_1^{\beta}$
                 $w^{\beta} \leftarrow \mathtt{BinSucc} \leftarrow z^0$                      $\mathtt{BinSucc} \subset_0 \mathtt{NumBits}$
               $\mid b1 \Rightarrow z^0 \leftarrow \mathtt{NumBits} \leftarrow x^{\alpha+1};$       $[0,0,-1,\infty]$   $z_1^0 = w_1^{\beta}$
                 $w^{\beta} \leftarrow \mathtt{BinSucc} \leftarrow z^0$                      $\mathtt{BinSucc} \subset_0 \mathtt{NumBits}$
               $\mid \$ \Rightarrow Rw^{\beta}.\mu_{\mathsf{bin}}; Rw^{\beta+1}.\$;\ w^{\beta+1} \leftarrow x^{\alpha+1}))$   $[0,0,-1,1]$
```

This version of the algorithm recognizes both recursive calls as valid. In the following definition we capture the idea of visibility from a type more formally. △

Definition 8. For type A and a set of type variables Δ, we define $\mathsf{c}(A;\Delta)$ inductively as:

$$\begin{aligned}
&\mathsf{c}(1;\Delta) = \emptyset,\\
&\mathsf{c}(\oplus\{\ell : A_\ell\}_{\ell\in L};\Delta) = \mathsf{c}(\&\{\ell : A_\ell\}_{\ell\in L};\Delta) = \textstyle\bigcup_{\ell\in L} \mathsf{c}(A_\ell;\Delta),\\
&\mathsf{c}(t;\Delta) = \{t\} \cup \mathsf{c}(A;\Delta\cup\{t\}) \text{ if } t =_a A \text{ and } t \notin \Delta,\\
&\mathsf{c}(t;\Delta) = \{t\} \text{ if } t =_a A \text{ and } t \in \Delta.
\end{aligned}$$

We put priority i in the set $\mathsf{c}(A)$ iff for some type variable t with $i = p(t)$, $t \in \mathsf{c}(A;\emptyset)$. Priority i is visible from type A if and only if $i \in \mathsf{c}(A)$.

9. Typing Rules for Session-Typed Processes with Channel Ordering

In this section we introduce inference rules for session-typed processes corresponding to derivations in subsingleton logic with fixed points. This is a refinement of the inference rules in Figure 2 to account for channel generations and orderings introduced in previous sections. The judgments are of the form

$$\bar{x}^{\alpha} : \omega \vdash_{\Omega} P :: (y^{\beta} : A),$$

where P is a process, and x^{α} (the α-th generation of channel x) and y^{β} (the β-th generation of channel y) are its left and right channels of types ω and A, respectively. The order relation between the generations of left and right channels indexed by their priority of types is built step by step in Ω when reading the rules from the conclusion to the premises. We only consider judgments in which all variables $x^{\alpha'}$ occurring in Ω are such that $\alpha' \leq \alpha$ and, similarly, for $y^{\beta'}$ in Ω we have $\beta' \leq \beta$. This presupposition guarantees that if we construct a derivation bottom-up, any future generations for x and y are fresh and not yet constrained by Ω. All our rules, again read bottom-up, will preserve this property.

We fix a signature Σ as in Definition 1, a finite set of process definitions V over Σ as in Definition 2, and define $\bar{x}^{\alpha} : \omega \vdash_{\Omega} P :: (y^{\beta} : A)$ with the rules in Figure 3. To preserve freshness of channels and their future generations in Ω, the channel introduced by Cut rule must be distinct from any variable mentioned in Ω. Similar to its underlying sequent calculus in Section 3, this system is infinitary, i.e., an infinite derivation may be produced for a given program. However, as long as the set of all process variables V is finite, we can map the infinite derivation of a finite program back into a circular pre-proof in the underlying sequent calculus. Programs derived in this system are all *type checked*, but not necessarily valid. It is, however, the basis on which we build our finite system of *(local) validity* in Section 10.

10. A Local Validity Condition

In Sections 4 to 7, using several examples, we developed an algorithm for identifying *valid* programs. Illustrating the full algorithm based on the inference rules in Section 9 was postponed to this section. We reserve it for the next section to prove our main result that the programs accepted by this algorithm satisfy the guard condition introduced by Fortier and Santocanale [FS13].

The condition checked by our algorithm is a *local* one in the sense that we check validity of each process definition in a program separately. The algorithm works on the sequents of the form

$$\langle \bar{u}^{\gamma}, X, v^{\delta} \rangle ; \bar{z}^{\alpha} : \omega \vdash_{\Omega, \subset} P :: (w^{\beta} : C),$$

where $\bar{u}^{\gamma}$ is the left channel of the process the algorithm started with and can be either empty or u^{γ}; it is empty if the process has no left channel. Similarly, v^{δ} is the right channel of the process the algorithm started with (that cannot be empty). And X is the last process variable a definition rule has applied to (reading the rules bottom-up). Again, in this judgment the (in)equalities in Ω can only relate variables z and w from earlier generations to guarantee freshness of later generations.

Generally speaking, when analysis of the program starts with $\bar{u}^{\gamma} : \omega \vdash v^{\delta} \leftarrow X \leftarrow \bar{u}^{\gamma} :: (v^{\delta} : B)$, a snapshot of the channels $\bar{u}^{\gamma}$ and v^{δ} and the process variable X are saved. Whenever the process reaches a call $\bar{z}^{\alpha} : \_ \vdash w^{\beta} \leftarrow Y \leftarrow \bar{z}^{\alpha} :: (w^{\beta} : \_)$, the algorithm compares $X, list(\bar{u}^{\gamma}, v^{\delta})$ and $Y, list(\bar{z}^{\alpha}, w^{\beta})$ using the $(\subset, <)$ order to determine if the recursive call is (locally) valid. This comparison is made by the Call rule in the rules in Figure 4.

Definition 9. A program $\mathcal{P} = \langle V, S \rangle$ over signature Σ and a fixed order $\subset$ satisfying properties in Definition 6, is *locally valid* iff for every $\bar{z} : A \vdash X = P_{\bar{z},w} :: (w : C) \in V$, there is a proof for

$$\langle \bar{z}^{0}, X, w^{0} \rangle ; \bar{z}^{0} : \omega \vdash_{\emptyset, \subset} P_{\bar{z}^{0},w^{0}} :: (w^{0} : C)$$

in the rule system in Figure 4. This set of rules is *finitary* so it can be directly interpreted as an algorithm. This results from substituting the Def rule with the Call rule. Again, to guarantee

$$\frac{}{x^\alpha : A \vdash_\Omega y^\beta \leftarrow x^\alpha :: (y^\beta : A)}\textsc{Id}$$

$$\mathtt{r}(v) = \{w^0_{p(s)} = v_{p(s)} \mid p(s) \notin \mathtt{c}(A)\}$$
$$\frac{\bar{x}^\alpha : \omega \vdash_{\Omega \cup \mathtt{r}(y^\beta)} P_{w^0} :: (w^0 : A) \qquad w^0 : A \vdash_{\Omega \cup \mathtt{r}(\bar{x}^\alpha)} Q_{w^0} :: (y^\beta : C)}{\bar{x}^\alpha : \omega \vdash_\Omega (w \leftarrow P_w; Q_w) :: (y^\beta : C)}\textsc{Cut}^w$$

$$\frac{\bar{x}^\alpha : \omega \vdash_\Omega P :: (y^\beta : A_k) \quad (k \in L)}{\bar{x}^\alpha : \omega \vdash_\Omega Ry^\beta.k; P :: (y^\beta : \oplus\{\ell : A_\ell\}_{\ell \in L})}\oplus R \qquad \frac{\forall \ell \in L \quad x^\alpha : A_\ell \vdash_\Omega P_\ell :: (y^\beta : C)}{x^\alpha : \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash_\Omega \mathbf{case}\, Lx^\alpha\ (\ell \Rightarrow P_\ell) :: (y^\beta : C)}\oplus L$$

$$\frac{\forall \ell \in L \quad \bar{x}^\alpha : \omega \vdash_\Omega P_\ell :: (y^\beta : A_\ell)}{\bar{x}^\alpha : \omega \vdash_\Omega \mathbf{case}\, Ry^\beta\ (\ell \Rightarrow P_\ell) :: (y^\beta : \&\{\ell : A_\ell\}_{\ell \in L})}\& R \qquad \frac{k \in L \quad x^\alpha : A_k \vdash_\Omega P :: (y^\beta : C)}{x^\alpha : \&\{\ell : A_l\}_{\ell \in L} \vdash_\Omega Lx^\alpha.k; P :: (y^\beta : C)}\& L$$

$$\frac{}{. \vdash_\Omega \mathbf{close}\, Ry^\beta :: (y^\beta : 1)}1R \qquad \frac{. \vdash_\Omega Q :: (y^\beta : A)}{x^\alpha : 1 \vdash_\Omega \mathbf{wait}\, Lx^\alpha; Q :: (y^\beta : A)}1L$$

$$\Omega' = \Omega \cup \{(y^\beta)_{p(s)} = (y^{\beta+1})_{p(s)} \mid p(s) \neq p(t)\}$$
$$\frac{\bar{x}^\alpha : \omega \vdash_{\Omega'} P_{y^{\beta+1}} :: (y^{\beta+1} : A) \qquad t =_\mu A}{\bar{x}^\alpha : \omega \vdash_\Omega Ry^\beta.\mu_t; P_{y^\beta} :: (y^\beta : t)}\mu R$$

$$\Omega' = \Omega \cup \{x^{\alpha+1}_{p(t)} < x^\alpha_{p(t)}\} \cup \{x^{\alpha+1}_{p(s)} = x^\alpha_{p(s)} \mid p(s) \neq p(t)\}$$
$$\frac{x^{\alpha+1} : A \vdash_{\Omega'} Q_{x^{\alpha+1}} :: (y^\beta : C) \qquad t =_\mu A}{x^\alpha : t \vdash_\Omega \mathbf{case}\, Lx^\alpha\ (\mu_t \Rightarrow Q_{x^\alpha}) :: (y^\beta : C)}\mu L$$

$$\Omega' = \Omega \cup \{y^{\beta+1}_{p(t)} < y^\beta_{p(t)}\} \cup \{y^{\beta+1}_{p(s)} = y^\beta_{p(s)} \mid p(s) \neq p(t)\}$$
$$\frac{\bar{x}^\alpha : \omega \vdash_{\Omega'} P_{y^{\beta+1}} :: (y^{\beta+1} : A) \qquad t =_\nu A}{\bar{x}^\alpha : \omega \vdash_\Omega \mathbf{case}\, Ry^\beta\ (\nu_t \Rightarrow P_{y^\beta}) :: (y^\beta : t)}\nu R$$

$$\Omega' = \Omega \cup \{(x^{\alpha+1})_{p(s)} = (x^\alpha)_{p(s)} \mid p(s) \neq p(t)\}$$
$$\frac{x^{\alpha+1} : A \vdash_{\Omega'} Q_{x^{\alpha+1}} :: (y^\beta : C) \qquad t =_\nu A}{x^\alpha : t \vdash_\Omega Lx^\alpha.\nu_t; Q_{x^\alpha} :: (y^\beta : C)}\nu L$$

$$\frac{\bar{x}^\alpha : \omega \vdash_\Omega P_{\bar{x}^\alpha, y^\beta} :: (y^\beta : C) \quad \bar{x} : \omega \vdash X = P_{\bar{x},y} :: (y : C) \in V}{\bar{x}^\alpha : \omega \vdash_\Omega y^\beta \leftarrow X \leftarrow \bar{x}^\alpha :: (y^\beta : C)}\textsc{Def}(X)$$

Figure 3: Infinitary Typing Rules for Processes with Channel Ordering

freshness of future generations of channels, the channel introduced by Cut rule is distinct from other variables mentioned in Ω.

The starting point of the algorithm can be of an arbitrary form

$$\langle \overline{z}^{\alpha}, X, w^{\beta}\rangle; \overline{z}^{\alpha} : \omega \vdash_{\Omega,\subset} P_{\overline{z}^{\alpha},w^{\beta}} :: (w^{\beta} : C),$$

as long as $\overline{z}^{\alpha+i}$ and $w^{\beta+i}$ do not occur in Ω for every $i > 0$. In both the inference rules and the algorithm, it is implicitly assumed that the next generation of channels introduced in the $\mu/\nu - R/L$ rules do not occur in Ω. Having this condition we can convert a proof for

$$\langle \overline{z}^{0}, X, w^{0}\rangle; \overline{z}^{0} : \omega \vdash_{\emptyset,\subset} P_{\overline{z}^{0},w^{0}} :: (w^{0} : C),$$

to a proof for

$$\langle \overline{z}^{\alpha}, X, w^{\beta}\rangle; \overline{z}^{\alpha} : \omega \vdash_{\Omega,\subset} P_{\overline{z}^{\alpha},w^{\beta}} :: (w^{\beta} : C),$$

by rewriting each $\overline{z}^{\gamma}$ and w^{δ} in the proof as $\overline{z}^{\gamma+\alpha}$ and $w^{\delta+\beta}$, respectively. This simple proposition is used in the next section where we prove that every locally valid process accepted by our algorithm is a valid proof according to the FS guard condition.

Proposition 4. If there is a deduction of

$$\langle \overline{z}^{0}, X, w^{0}\rangle; \overline{z}^{0} : \omega \vdash_{\emptyset,\subset} P_{\overline{z}^{0},w^{0}} :: (w^{0} : C),$$

then there is also a deduction of

$$\langle \overline{z}^{\alpha}, X, w^{\beta}\rangle; \overline{z}^{\alpha} : \omega \vdash_{\Omega,\subset} P_{\overline{z}^{\alpha},w^{\beta}} :: (w^{\beta} : C),$$

if for all $0 < i$, $\overline{z}^{\alpha+i}$ and $w^{\beta+i}$ do not occur in Ω.

Proof. By substitution, as explained above. □

11. Local Validity and Guard Conditions

Fortier and Santocanale [FS13] introduced a *guard condition* for identifying valid circular proofs among all infinite pre-proofs in the singleton logic with fixed points. They showed that the pre-proofs satisfying this condition, which is based on the definition of left $\mu-$ and right $\nu-$ traces, enjoy the cut elimination property. In this section, we translate their guard condition into the context of session-typed concurrency and generalize it for subsingleton logic. It is immediate that the cut elimination property holds for a proof in subsingleton logic if it satisfies the generalized version of the guard condition. The key idea is that cut reductions for individual rules stay untouched in subsingleton logic and rules for the new constant 1 only provide more options for a proof in this system to terminate. We prove that all locally valid programs in the session typed system, determined by the algorithm in Section 10, also satisfy the guard condition. We conclude that our algorithm imposes a stricter but local version of validity on the session-typed programs corresponding to the circular pre-proofs.

Here we adapt definitions of the *left* and *right traceable* paths, *left* μ- and *right* ν-*traces,* and then *validity* to our session type system.

Definition 10. Consider path $\mathbb{P}$ on a program $Q = \langle V, S\rangle$ defined on a Signature Σ:

$$\cfrac{\cfrac{\overline{x}^{\gamma} : \omega' \vdash_{\Omega'} Q' :: (y^{\delta} : C')}{\vdots}}{\overline{z}^{\alpha} : \omega \vdash_{\Omega} Q :: (w^{\beta} : C)}$$

$$\frac{}{\langle \bar{u}^\gamma, X, v^\delta\rangle; z^\alpha : A \vdash_{\Omega,\subset} w^\beta \leftarrow z^\alpha :: (w^\beta : A)}\,\textsc{Id}$$

$$\frac{\begin{array}{c}\mathtt{r}(y) = \{w^0_{p(s)} = y_{p(s)} \mid p(s) \notin \mathtt{c}(A)\}\\ \langle \bar{u}^\gamma, X, v^\delta\rangle; \bar{z}^\alpha : \omega \vdash_{\Omega\cup\mathtt{r}(w^\beta),\subset} P_{x^0} :: (x^0 : A) \quad \langle \bar{u}^\gamma, X, v^\delta\rangle; x^0 : A \vdash_{\Omega\cup\mathtt{r}(\bar{z}^\alpha),\subset} Q_{x^0} :: (w^\beta : C)\end{array}}{\langle \bar{u}^\gamma, X, v^\delta\rangle; \bar{z}^\alpha : \omega \vdash_{\Omega\subset} (x \leftarrow P_x; Q_x) :: (w^\beta : C)}\,\textsc{Cut}^x$$

$$\frac{\langle \bar{u}^\gamma, X, v^\delta\rangle; \bar{z}^\alpha : \omega \vdash_{\Omega,\subset} P :: (w^\beta : A_k) \quad (k \in L)}{\langle \bar{u}^\gamma, X, v^\delta\rangle; \bar{z}^\alpha : \omega \vdash_{\Omega,\subset} Rw^\beta.k; P :: (w^\beta : \oplus\{\ell : A_l\}_{\ell\in L})}\,\oplus R$$

$$\frac{\forall \ell \in L \quad \langle \bar{u}^\gamma, X, v^\delta\rangle; z^\alpha : A_\ell \vdash_{\Omega,\subset} P_\ell :: (w^\beta : C)}{\langle \bar{u}^\gamma, X, v^\delta\rangle; z^\alpha : \oplus\{\ell : A\}_{\ell\in L} \vdash_{\Omega,\subset} \mathbf{case}\, Lz^\alpha\, (\ell \Rightarrow P_\ell) :: (w^\beta : C)}\,\oplus L$$

$$\frac{\forall \ell \in L \quad \langle \bar{u}^\gamma, X, v^\delta\rangle; \bar{z}^\alpha : \omega \vdash_{\Omega,\subset} P_\ell :: (w^\beta : A_\ell)}{\langle \bar{u}^\gamma, X, v^\delta\rangle; \bar{z}^\alpha : \omega \vdash_{\Omega,\subset} \mathbf{case}\, Rw^\beta\, (\ell \Rightarrow P_\ell) :: (w^\beta : \&\{\ell : A_\ell\}_{\ell\in L})}\,\& R$$

$$\frac{(k \in L) \quad \langle \bar{u}^\gamma, X, v^\delta\rangle; z^\alpha : A_k \vdash_{\Omega,\subset} P :: (w^\beta : C)}{\langle \bar{u}^\gamma, X, v^\delta\rangle; z^\alpha : \&\{\ell : A_\ell\}_{\ell\in L} \vdash_{\Omega,\subset} Lz^\alpha.k; P :: (w^\beta : C)}\,\& L$$

$$\frac{}{\langle \bar{u}^\gamma, X, v^\delta\rangle; \cdot \vdash_{\Omega,\subset} \mathbf{close}\, R :: (w^\beta : 1)}\,1R \qquad \frac{\langle \bar{u}^\gamma, X, v^\delta\rangle; \cdot \vdash_{\Omega,\subset} Q :: (w^\beta : A)}{\langle \bar{u}^\gamma, X, v^\delta\rangle; z^\alpha : 1 \vdash_{\Omega,\subset} \mathbf{wait}\, Lz^\alpha; Q :: (w^\beta : A)}\,1L$$

$$\frac{\begin{array}{c}\Omega' = \Omega \cup \{w^\beta_{p(s)} = w^{\beta+1}_{p(s)} \mid p(s) \neq p(t)\}\\ \langle \bar{u}^\gamma, X, v^\delta\rangle; \bar{z}^\alpha : \omega \vdash_{\Omega',\subset} P_{w^{\beta+1}} :: (w^{\beta+1} : A) \quad t =_\mu A\end{array}}{\langle \bar{u}^\gamma, X, v^\delta\rangle; \bar{z}^\alpha : \omega \vdash_{\Omega,\subset} Rw^\beta.\mu_t; P_{w^\beta} :: (w^\beta : t)}\,\mu R$$

$$\frac{\begin{array}{c}\Omega' = \Omega \cup \{z^{\alpha+1}_{p(t)} < z^\alpha_{p(t)}\} \cup \{z^{\alpha+1}_{p(s)} = z^\alpha_{p(s)} \mid p(s) \neq p(t)\}\\ \langle \bar{u}^\gamma, X, v^\delta\rangle; z^{\alpha+1} : A \vdash_{\Omega',\subset} Q_{z^{\alpha+1}} : (w^\beta :: C) \qquad t =_\mu A\end{array}}{\langle \bar{u}^\gamma, X, v^\delta\rangle; z^\alpha : t \vdash_{\Omega,\subset} \mathbf{case}\, Lz^\alpha\, (\mu_t \Rightarrow Q_{z^\alpha}) :: (w^\beta : C)}\,\mu L$$

$$\frac{\begin{array}{c}\Omega' = \Omega \cup \{w^{\beta+1}_{p(t)} < w^\beta_{p(t)}\} \cup \{w^{\beta+1}_{p(s)} = w^\beta_{p(s)} \mid p(s) \neq p(t)\}\\ \langle \bar{u}^\gamma, X, v^\delta\rangle; \bar{z}^\alpha : \omega \vdash_{\Omega',\subset} P_{w^{\beta+1}} :: (w^{\beta+1} : A) \qquad t =_\nu A\end{array}}{\langle \bar{u}^\gamma, X, v^\delta\rangle; \bar{z}^\alpha : \omega \vdash_{\Omega,\subset} \mathbf{case}\, Rw^\beta\, (\nu_t \Rightarrow P_{w^\beta}) :: (w^\beta : t)}\,\nu R$$

$$\frac{\begin{array}{c}\Omega' = \Omega \cup \{z^{\alpha+1}_{p(s)} = z^\alpha_{p(s)} \mid p(s) \neq p(t)\}\\ \langle \bar{u}^\gamma, X, v^\delta\rangle; z^{\alpha+1} : A \vdash_{\Omega',\subset} Q_{z^{\alpha+1}} :: (w^\beta : C) \quad t =_\nu A\end{array}}{\langle \bar{u}^\gamma, X, v^\delta\rangle; z^\alpha : t \vdash_{\Omega,\subset} Lz^\alpha.\nu_t; Q_{z^\alpha} :: (w^\beta : C)}\,\nu L$$

$$\frac{Y, list(\bar{z}^\alpha, w^\beta)\ (\subset, <_\Omega)\ X, list(\bar{u}^\gamma, v^\delta) \quad \bar{x} : \omega \vdash Y = P_{\bar{x},y} :: (y : C) \in V}{\langle \bar{u}^\gamma, X, v^\delta\rangle; \bar{z}^\alpha : \omega \vdash_{\Omega,\subset} w^\beta \leftarrow Y \leftarrow \bar{z}^\alpha :: (w^\beta : C)}\,\textsc{Call}$$

Figure 4: Finitary Rules for Local Validity

$\mathbb{P}$ is called *left traceable* if $\bar{z}$ and $\bar{x}$ are non-empty and $\bar{z} = \bar{x}$. It is called *right traceable* if $w = y$.

Definition 11. A path $\mathbb{P}$ on a program $Q = \langle V, S \rangle$ defined over Signature Σ is a left μ-trace if (i) it is left-traceable, (ii) there is a left fixed point rule applied on it, and (iii) the highest priority of its left fixed point rule is $i \leq n$ such that $\epsilon(i) = \mu$. Dually, $\mathbb{P}$ is a right ν-trace if (i) it is right-traceable, (ii) there is a right fixed point rule applied on it, and (iii) the highest priority of its right fixed point is $i \leq n$ such that $\epsilon(i) = \nu$.

Definition 12 (FS guard condition on cycles)**.** A program $Q = \langle V, S \rangle$ defined on Signature Σ satisfies the FS guard condition if every cycle $\mathbb{C}$

$$\dfrac{\dfrac{\bar{x}^{\gamma} : \omega' \vdash_{\Omega'} y^{\delta} \leftarrow X \leftarrow \bar{x}^{\gamma} :: (y^{\delta} : C')}{\vdots}}{\bar{z}^{\alpha} : \omega \vdash_{\Omega} w^{\beta} \leftarrow X \leftarrow \bar{z}^{\alpha} :: (w^{\beta} : C)}$$

over Q is either a left μ-trace or a right ν-trace. Similarly, we say a single cycle $\mathbb{C}$ satisfies the guard condition if it is either a left μ-trace or a right ν-trace.

Definitions 10-12 are equivalent to the definitions of the same concepts by Fortier and Santocanale using our own notation. As an example, consider program $\mathcal{P}_3 := \langle \{\mathsf{Copy}\}, \mathsf{Copy} \rangle$ over signature Σ_1, defined in Example 6, where Copy has types $x : \mathsf{nat} \vdash \mathsf{Copy} :: (y : \mathsf{nat})$.

$$\Sigma_1 := \mathsf{nat} =^1_{\mu} \oplus\{z : 1, s : \mathsf{nat}\}$$

$$\begin{aligned} y \leftarrow \mathsf{Copy} \leftarrow x = \mathbf{case}\, Lx\, (\mu_{nat} \Rightarrow \mathbf{case}\, Lx\, (\, & z \Rightarrow Ry.\mu_{nat}; Ry.z; \mathbf{wait}\, Lx; \mathbf{close}\, Ry \\ & \mid s \Rightarrow Ry.\mu_{nat}; Ry.s; y \leftarrow \mathsf{Copy} \leftarrow x)) \end{aligned}$$

Consider the first several steps of the derivation of the program starting with $x^0 : \mathsf{nat} \vdash_{\emptyset} y^0 \leftarrow \mathsf{Copy} \leftarrow x^0 :: (y^0 : \mathsf{nat})$:

$$\dfrac{\dfrac{\dfrac{x^1 : 1 \vdash_{\{x_1^1 < x_1^0\}} Ry^0.\mu_{nat}; \cdots :: (y^0 : \mathsf{nat}) \qquad \dfrac{\dfrac{x^1 : \mathsf{nat} \vdash_{\{x_1^1 < x_1^0\}} y^1 \leftarrow \mathsf{Copy} \leftarrow x^1 :: (y^1 : \mathsf{nat})}{x^1 : \mathsf{nat} \vdash_{\{x_1^1 < x_1^0\}} Ry^1.s; \cdots :: (y^1 : 1 \oplus \mathsf{nat})}\oplus R}{x^1 : \mathsf{nat} \vdash_{\{x_1^1 < x_1^0\}} Ry^0.\mu_{nat}; \cdots :: (y^0 : \mathsf{nat})}\mu R}{x^1 : 1 \oplus \mathsf{nat} \vdash_{\{x_1^1 < x_1^0\}} \mathbf{case}\, Lx^1\, (\cdots) :: (y^0 : \mathsf{nat})}\oplus L}{x^0 : \mathsf{nat} \vdash_{\emptyset} \mathbf{case}\, Lx^0\, (\mu_{nat} \Rightarrow \cdots) :: (y^0 : \mathsf{nat})}\mu L}{x^0 : \mathsf{nat} \vdash_{\emptyset} y^0 \leftarrow \mathsf{Copy} \leftarrow x^0 :: (y^0 : \mathsf{nat})}\textsc{Def}(\mathsf{Copy})$$

The path between

$$x^0 : \mathsf{nat} \vdash_{\emptyset} y^0 \leftarrow \mathsf{Copy} \leftarrow x^0 :: (y^0 : \mathsf{nat})$$

and

$$x^1 : \mathsf{nat} \vdash_{\{x_1^1 < x_1^0\}} y^1 \leftarrow \mathsf{Copy} \leftarrow x^1 :: (y^1 : \mathsf{nat})$$

is by definition both left traceable and right traceable, but it is only a left μ-trace and not a right ν-trace: The highest priority of a fixed point applied on the left-hand side on this path belongs to a positive type; this application of the μL rule added $x_1^1 < x_1^0$ to the set defining the $<$ order. However, there is no negative fixed point rule applied on the right, and y_1^1 and y_1^0 are incomparable to each other.

This cycle, thus, satisfies the *guard condition* by being a left μ-trace, and it is also accepted by our algorithm since $list(x^1, y^1) = [(x_1^1, y_1^1)] < [(x_1^0, y_1^0)] = list(x^0, y^0)$, as being checked in the CALL rule.

Here, we can observe that being a left μ-trace coincides with having the relation $x_1^1 < x_1^0$ between the left channels, and not being a right ν-trace coincides with not having the relation $y_1^1 < y_1^0$ for the right channels. We can generalize this observation to every path and every signature with n priorities.

Theorem 5. A cycle $\mathbb{C}$

$$\cfrac{\cfrac{\bar{x}^{\gamma} : \omega' \vdash_{\Omega'} y^{\delta} \leftarrow X \leftarrow \bar{x}^{\gamma} :: (y^{\delta} : C')}{\vdots}}{\bar{z}^{\alpha} : \omega \vdash_{\Omega} w^{\beta} \leftarrow X \leftarrow \bar{z}^{\alpha} :: (w^{\beta} : C)}$$

on a program $Q = \langle V, S \rangle$ defined over Signature Σ is a *left μ-trace* if $\bar{x}$ and $\bar{z}$ are non-empty and the list $[x^{\gamma}] = [x_1^{\gamma}, \cdots, x_n^{\gamma}]$ is lexicographically less than the list $[z^{\alpha}] = [z_1^{\alpha}, \cdots, z_n^{\alpha}]$ by the order $<_{\Omega'}$ built in Ω'. Dually, it is a *right ν-trace*, if the list $[y^{\delta}] = [y_1^{\delta}, \cdots, y_n^{\delta}]$ is lexicographically less than the list $[w^{\beta}] = [w_1^{\beta}, \cdots, w_n^{\beta}]$ by the strict order $<_{\Omega'}$ built in Ω'

Proof. This theorem is a corollary of Lemmas 10 and 11 proved in Appendix A. □

We provide a few additional examples to elaborate Theorem 5 further. Define a new program $\mathcal{P}_9 := \langle\{\mathsf{Succ}, \mathsf{Copy}, \mathsf{SuccCopy}\}, \mathsf{SuccCopy}\rangle$, over the signature Σ_1, using the process $w : \mathsf{nat} \vdash \mathsf{Copy} :: (y : \mathsf{nat})$ and two other processs: $x : \mathsf{nat} \vdash \mathsf{Succ} :: (w : \mathsf{nat})$ and $x : \mathsf{nat} \vdash \mathsf{SuccCopy} :: (y : \mathsf{nat})$. The processes are defined as

$$w \leftarrow \mathsf{Succ} \leftarrow x = Rw.\mu_{nat}; Rw.s; w \leftarrow x$$

$$\begin{aligned} y \leftarrow \mathsf{Copy} \leftarrow w = \textbf{case}\, Lw\, (\mu_{nat} \Rightarrow \textbf{case}\, Lw\, (\, & s \Rightarrow Ry.\mu_{nat}; Ry.s; y \leftarrow \mathsf{Copy} \leftarrow w \\ \mid\, & z \Rightarrow Ry.\mu_{nat}; Ry.z; \textbf{wait}\, Lw; \textbf{close}\, Ry)) \end{aligned}$$

$$y \leftarrow \mathsf{SuccCopy} \leftarrow x = w \leftarrow \mathsf{Succ} \leftarrow x; y \leftarrow \mathsf{Copy} \leftarrow w,$$

Process SuccCopy spawns a new process Succ and continues as Copy. The Succ process prepends an s label to the beginning of the finite string representing a natural number on its left hand side and then forwards the string as a whole to the right. Copy receives this finite string, representing a natural number, on its left hand side, one element at a time, and *distributes* it to the right element by element. The only recursive process in this program is Copy that is discussed earlier in this section. So program $\mathcal{P}_9$, itself, does not have a further interesting point to discuss. We consider a bogus version of this program in Example 16 that provides further intuition for Theorem 5.

Example 16. Define program $\mathcal{P}_{10} := \langle\{\mathsf{Succ}, \mathsf{BogusCopy}, \mathsf{SuccCopy}\}, \mathsf{SuccCopy}\rangle$ over the signature

$$\Sigma_1 := \mathsf{nat} =_{\mu}^{1} \oplus\{z : 1,\ \ s : \mathsf{nat}\},$$

The processes $x : \mathsf{nat} \vdash \mathsf{Succ} :: (w : \mathsf{nat})$, $w : \mathsf{nat} \vdash \mathsf{BogusCopy} :: (y : \mathsf{nat})$, and $x : \mathsf{nat} \vdash \mathsf{SuccCopy} :: (y : \mathsf{nat})$, are defined as

$$w \leftarrow \mathsf{Succ} \leftarrow x = Rw.\mu_{nat}; Rw.s; w \leftarrow x$$

$$
\begin{aligned}
y \leftarrow \mathtt{BogusCopy} \leftarrow w = \mathbf{case}\, Lw\, (\mu_{nat} \Rightarrow \mathbf{case}\, Lw\, (\, & s \Rightarrow Ry.\mu_{nat}; Ry.s; y \leftarrow \mathtt{SuccCopy} \leftarrow w \\
\mid\, & z \Rightarrow Ry.\mu_{nat}; Ry.z; \mathbf{wait}\, Lw; \mathbf{close}\, Ry))
\end{aligned}
$$

$$y \leftarrow \mathtt{SuccCopy} \leftarrow x = w \leftarrow \mathtt{Succ} \leftarrow x; y \leftarrow \mathtt{BogusCopy} \leftarrow w$$

Program $\mathcal{P}_{10}$ is a non-reactive *bogus* program, since BogusCopy instead of calling itself recursively, calls SuccCopy. At the very beginning SuccCopy spawns Succ and continues with BogusCopy for a fresh channel w. Succ then sends a fixed point unfolding message and a *successor* label via w to the right, while BogusCopy receives the two messages just sent by Succ through w and calls SuccCopy recursively again. This loop continues forever, without any messages being received from the outside.

The first several steps of the derivation of $x^0 : \mathsf{nat} \vdash_\emptyset \mathtt{SuccCopy} :: (y^0 : \mathsf{nat})$ in our inference system (Section 9) are given below.

$$
\dfrac{
\dfrac{\dfrac{\dfrac{x^0 : \mathsf{nat} \vdash_\emptyset w^1 \leftarrow \mathtt{Succ} \leftarrow x^0 :: (w^1 : \mathsf{nat})}{x^0 : \mathsf{nat} \vdash_\emptyset Rw^1.z; \cdots :: (w^1 : 1 \oplus \mathsf{nat})}\oplus R}{x^0 : \mathsf{nat} \vdash_\emptyset Rw^0.\mu_{nat}; \cdots :: (w^0 : \mathsf{nat})}\mu R}{x^0 : \mathsf{nat} \vdash_\emptyset w^0 \leftarrow \mathtt{Succ} \leftarrow x^0 :: (w^0 : \mathsf{nat})}\textsc{Def}
\quad
\dfrac{\dfrac{\dfrac{\cdots \quad w^1 : \mathsf{nat} \vdash_{\{w^1_1 < w^0_1\}} y^0 \leftarrow \mathtt{SuccCopy} \leftarrow w^1 :: (y^0 : \mathsf{nat})}{w^1 : 1 \oplus \mathsf{nat} \vdash_{\{w^1_1 < w^0_1\}} \mathbf{case}\, Lw^1\, (\cdots) :: (y^0 : \mathsf{nat})}\oplus L}{w^0 : \mathsf{nat} \vdash_\emptyset \mathbf{case}\, Lw^0\, (\mu_{nat} \Rightarrow \cdots) :: (y^0 : \mathsf{nat})}\mu L}{w^0 : \mathsf{nat} \vdash_\emptyset y^0 \leftarrow \mathtt{BogusCopy} \leftarrow w^0 :: (y^0 : \mathsf{nat})}\textsc{Def}
}{
\dfrac{x^0 : \mathsf{nat} \vdash_\emptyset w \leftarrow \mathtt{Succ}; y^0 \leftarrow \mathtt{BogusCopy} \leftarrow w :: (y^0 : \mathsf{nat})}{x^0 : \mathsf{nat} \vdash_\emptyset y^0 \leftarrow \mathtt{SuccCopy} \leftarrow x^0 :: (y^0 : \mathsf{nat})}\textsc{Def}
}\textsc{Cut}^w
$$

Consider the cycle between

$$x^0 : \mathsf{nat} \vdash_\emptyset y^0 \leftarrow \mathtt{SuccCopy} \leftarrow x^0 :: (y^0 : \mathsf{nat})$$

and

$$w^1 : \mathsf{nat} \vdash_\emptyset y^0 \leftarrow \mathtt{SuccCopy} \leftarrow w^1 :: (y^0 : \mathsf{nat}).$$

By Definition 11, this path is right traceable, but not left traceable. And by Definition 10, the path is neither a right ν-trace nor a left μ-trace:

(1) No negative fixed point unfolding message is received from the right and y^0 does not evolve to a new generation that has a smaller value in its highest priority than y^0_1. In other words, $y^0_1 \not< y^0_1$ since no negative fixed point rule has been applied on the right channel.

(2) The positive fixed point unfolding message that is received from the left is received through the channel w^0, which is a fresh channel created after SuccCopy spawns the process Succ. Although $w^1_1 < w^0_1$, since x^0_1 is incomparable to w^0_1, the relation $w^1_1 < x^0_1$ does not hold. This path is not even a left-traceable path.

Neither $[w^1] = [w^1_1] < [x^0_1] = [x^0]$, nor $[y^0] = [y^0_1] < [y^0_1] = [y^0]$ hold, and this cycle does not satisfy the guard condition. This program is not locally valid either since $[w^1_1, y^0_1] \not< [x^0_1, y^0_1]$. △

As another example consider the program $\mathcal{P}_6 = \langle\{\mathtt{Ping}, \mathtt{Pong}, \mathtt{PingPong}\}, \mathtt{PingPong}\rangle$ over the signature Σ_4 as defined in Example 9. We discussed in Section 6 that this program is not accepted by our algorithm as locally valid.

Example 17.

$$\begin{aligned}\Sigma_4 := \mathsf{ack} &=^1_\mu \oplus\{ack : \mathsf{astream}\},\\ \mathsf{astream} &=^2_\nu \&\{head : \mathsf{ack},\ \ tail : \mathsf{astream}\},\\ \mathsf{nat} &=^3_\mu \oplus\{z : 1,\ \ s : \mathsf{nat}\}\end{aligned}$$

Processes

$$\begin{aligned}&x : \mathsf{nat} \vdash \texttt{Ping} :: (w : \mathsf{astream}),\\ &w : \mathsf{astream} \vdash \texttt{Pong} :: (y : \mathsf{nat}),\\ &x : \mathsf{nat} \vdash \texttt{PingPong} :: (y : \mathsf{nat})\end{aligned}$$

are defined as

$$\begin{aligned}w \leftarrow \texttt{Ping} \leftarrow x = \mathbf{case}\, Rw\, (\nu_{astream} \Rightarrow \mathbf{case}\, Rw\, (\ &head \Rightarrow Rw.\mu_{ack}; Rw.ack; w \leftarrow \texttt{Ping} \leftarrow x\\ \mid\ &tail \Rightarrow w \leftarrow \texttt{Ping} \leftarrow x))\end{aligned}$$

$$\begin{aligned}y \leftarrow \texttt{Pong} \leftarrow w =& Lw.\nu_{astream}; Lw.head;\\ &\mathbf{case}\, Lw\, (\mu_{ack} \Rightarrow \mathbf{case}\, Lw\, (ack \Rightarrow Ry.\mu_{nat}; Ry.s; y \leftarrow \texttt{Pong} \leftarrow w))\end{aligned}$$

$$y \leftarrow \texttt{PingPong} \leftarrow x = w \leftarrow \texttt{Ping} \leftarrow x; y \leftarrow \texttt{Pong} \leftarrow w$$

The first several steps of the proof of $x^0 : nat \vdash_\emptyset \texttt{PingPong} :: (y^0 : nat)$ in our inference system (Section 9) are given below (with some abbreviations).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x^0 : \mathsf{nat} \vdash_B w^2 \leftarrow \texttt{Ping} \leftarrow x^0 :: (w^2 : \mathsf{astream})}{x^0 : \mathsf{nat} \vdash_B Rw^2.ack; \cdots :: (w^2 : \oplus\{\mathsf{astream}\})}\oplus R}{x^0 : \mathsf{nat} \vdash_A Rw^1.\mu_{ack}; \cdots :: (w^1 : \mathsf{ack})}\mu R \qquad x^0 : \mathsf{nat} \vdash_A \cdots :: (w^1 : \mathsf{astream})}{x^0 : \mathsf{nat} \vdash_A \mathbf{case}\, Rw^1\, (\cdots) :: (w^1 : \mathsf{ack} \,\&\, \mathsf{astream})}\&R}{x^0 : \mathsf{nat} \vdash_\emptyset \mathbf{case}\, Rw^0\, (\nu_{astream} \Rightarrow \cdots) :: (w^0 : \mathsf{astream})}\nu R}{x^0 : \mathsf{nat} \vdash_\emptyset w^0 \leftarrow \texttt{Ping} \leftarrow x^0 :: (w^0 : \mathsf{astream})}\textsc{Def} \qquad w^0 : \mathsf{astream} \vdash_\emptyset \cdots :: (y^0 : \mathsf{nat})}{x^0 : \mathsf{nat} \vdash_\emptyset w \leftarrow \texttt{Ping} \leftarrow x^0; y^0 \leftarrow \texttt{Pong} \leftarrow w :: (y^0 : \mathsf{nat})}\textsc{Cut}}{x^0 : \mathsf{nat} \vdash_\emptyset y^0 \leftarrow \texttt{PingPong} \leftarrow x^0 :: (y^0 : \mathsf{nat})}\textsc{Def}$$

where $A = \{w^1_1 = w^0_1, w^1_2 < w^0_2, w^1_3 = w^0_3\}$, and $B = \{w^2_1 = w^1_1 = w^0_1, w^1_2 < w^0_2, w^2_3 = w^1_3 = w^0_3, \}$. The cycle between the processes

$$x^0 : \mathsf{nat} \vdash_\emptyset w^0 \leftarrow \texttt{Ping} \leftarrow x^0 :: (w^0 : \mathsf{astream})$$

and

$$x^0 : \mathsf{nat} \vdash_B w^2 \leftarrow \texttt{Ping} \leftarrow x^0 :: (w^2 : \mathsf{astream})$$

is neither a left μ-trace, nor a right ν-trace:

(1) No fixed point unfolding message is received or sent through the left channels in this path and thus $[x^0] = [x^0_1, x^0_2, x^0_3] \not< [x^0_1, x^0_2, x^0_3] = [x^0]$.
(2) On the right, fixed point unfolding messages are both sent and received: (i) w^0 receives an unfolding message for a negative fixed point with priority 2 and evolves to w^1, and then later (ii) w^1 sends an unfolding message for a positive fixed point with priority 1 and evolves to w^2. But the positive fixed point has a higher priority than the negative fixed point, and thus this path is not a right ν-trace either.

This reasoning can also be reflected in our observation about the list of channels in Theorem 5: When, first, w^0 evolves to w^1 by receiving a message in (i) the relations $w^0_1 = w^1_1$, $w^0_2 < w^1_2$, and $w^0_3 = w^1_3$ are recorded. And, later, when w^1 evolves to w^2 by sending a message in (ii) the relations $w^1_2 = w^2_2$, and $w^1_3 = w^2_3$ are added to the set. This means that w^2_1 as the first element of the list $[w^2]$ remains incomparable to w^0_1 and thus $[w^2] = [w^2_1, w^2_2, w^2_3] \not< [w^0_1, w^0_2, w^0_3] = [w^0]$ △

We are now ready to state our main theorem that connects the local validity algorithm introduced in Section 10 to the FS guard condition.

Theorem 6. A locally valid program satisfies the FS guard condition.

Proof. We give a sketch of the proof here. See Appendix A for the complete proof.

By Theorem 5, a cycle $\mathbb{C}$

$$\cfrac{\cfrac{\bar{x}^\gamma : \omega' \vdash_{\Omega'} y^\delta \leftarrow X \leftarrow \bar{x}^\gamma :: (y^\delta : C')}{\vdots}}{\bar{z}^\alpha : \omega \vdash_{\Omega} w^\beta \leftarrow X \leftarrow \bar{z}^\alpha :: (w^\beta : C)}$$

is either a *left μ-trace* or a *right ν-trace* if either $[x^\gamma] <_{\Omega'} [z^\alpha]$ **or** $[y^\delta] <_{\Omega'} [w^\beta]$ holds. The FS validity condition requires this disjunctive condition for all cycles in the program. In our algorithm, however, we merge the lists of left and right channels, e.g. $[x^\gamma]$ and $[y^\delta]$ respectively, into a single list $list(x^\gamma, y^\delta)$. The values in $list(x^\gamma, y^\delta)$ from Definition 4 are still recorded in their order of priorities, but for the same priority the value corresponding to *receiving* a message precedes the one corresponding to *sending* a message. Roughly, but not exactly, instead of checking $[x^\gamma] <_{\Omega'} [z^\alpha]$ or $[y^\delta] <_{\Omega'} [w^\beta]$, our algorithm checks $list(x^\gamma, y^\delta) <_{\Omega'} list(z^\alpha, w^\beta)$.

To make it exact we should also note that our algorithm checks all calls even those that do not form a cycle in the sense of the FS guard condition (that is, when process X calls process $Y \neq X$). By adding process variables to our validity check condition, as described in Definition 7, there is no need to search for every possible cycle in the program. Instead, our algorithm only checks the condition for the immediate recursive calls that a process makes. As this condition enjoys transitivity, it also holds for all possible non-immediate recursive calls, including the cycles. □

Note that checking a disjunctive condition for each cycle implies that we cannot rely on transitivity. Therefore, to check the FS guard condition we have to examine every possible cycle separately, even if it is composed of two previously checked cycles. Our local validity condition, however, works on a single transitive condition based on a single list of channels and one process variable. By merging the lists and checking the condition only for the immediate recursive calls, our algorithm is *local* but *stricter* than the FS guard condition.

12. Computational Meta-theory

Fortier and Santocanale [FS13] defined a function Treat as a part of their cut elimination algorithm. They proved that this function terminates on a list of pre-proofs fused by consecutive cuts if all of them satisfy their guard condition. In our system, function Treat corresponds to computation on a configuration of processes. In this section we first show that usual preservation and progress theorems hold even if a program does not satisfy the validity condition. Then we use Fortier and Santocanale's result to prove a stronger compositional progress property for (locally) valid programs.

In Section 3, we introduced process configurations C as a list of processes connected by the associative, noncommutative parallel composition operator $|_x$.

$$C ::= \cdot \mid \mathrm{P} \mid (C_1 \;|_x\; C_2)$$

with unit $(\cdot)$. The type checking judgements for configurations $\bar{x} : \omega \Vdash C :: (y : B)$ are:

$$\frac{}{x : A \Vdash \cdot :: (x : A)} \qquad \frac{\bar{x} : \omega \vdash P :: (y : B)}{\bar{x} : \omega \Vdash P :: (y : B)} \qquad \frac{\bar{x} : \omega \Vdash C_1 :: (z : A) \quad z : A \Vdash C_2 :: (y : B)}{\bar{x} : \omega \Vdash C_1|_z\, C_2 :: (y : B)}$$

A configuration can be read as a list of processes connected by consecutive cuts. Alternatively, considering C_1 and C_2 as two processes, configuration $C_1 \;|_z\; C_2$ can be read as their composition by a cut rule $(z \leftarrow C_1; C_2)$. In section 3, we defined an operational semantics on configurations using transition rules. Similarly, these computational transitions can be interpreted as cut reductions called "internal operations" by Fortier and Santocanale . The usual preservation theorem ensures types of a configuration are preserved during computation [DP16].

Theorem 7. (Preservation) For a configuration $\bar{x} : \omega \Vdash C :: (y : A)$, if $C \mapsto C'$ by one step of computation, then $\bar{u} : \omega \Vdash C' : (y : A)$.

Proof. This property follows directly from the correctness of cut reduction steps. □

The usual progress property as defined below ensures that computation makes progress or it attempts to communicate with an external process.

Theorem 8. (Progress) Configuration $\bar{x} : \omega \Vdash C :: (y : A)$ satisfies *progress* if

(1) either C can make a transition,
(2) or $C = (\cdot)$ is empty,
(3) or C attempts to communicate either to the left or to the right.

Proof. The proof is by structural induction on the configuration typing. □

In the presence of (mutual) recursion, this progress property is not strong enough to ensure that a program does not get stuck in an infinite inner loop. Since our local validity condition implies the FS guard condition, we can use their results for a stronger version of the progress theorem on valid programs.

Theorem 9. (Strong Progress) Configuration $\bar{x} : \omega \Vdash C :: (y : A)$ of (locally) valid processes satisfies the progress property. Furthermore, after a finite number of steps, either

(1) $C = (\cdot)$ is empty,
(2) or C attempts to communicate to the left or right.

Proof. There is a correspondence between Treat function's internal operations and computational transitions introduced in Section 3. The only point of difference is the extra computation rule we introduced for constant 1. Fortier and Santocanale's proof for termination of cut elimination remains intact after extending Treat's primitive operation with a reduction rule for constant 1, since this reduction step only introduces a new way of closing a process in the configuration. Under this correspondence, termination of the function Treat on valid proofs implies the strong progress property for valid programs. □

As a corollary to Theorem 9, computation of a closed valid program $\mathcal{P} = \langle V, S \rangle$ with $\cdot \vdash S = P :: (y : 1)$ always terminates by closing the channel y (which follows by inversion on the typing derivation).

We conclude this section by briefly revisiting sources of invalidity in computation. In Example 4 we saw that process Loop is not valid, even though its proof is cut-free. Its computation satisfies the strong progress property as it attempts to communicate with its right side in finite number of steps. However, its communication with left and right sides of the configuration is solely by sending messages. Composing Loop with any process $y : \mathsf{nat} \vdash \mathtt{P} :: (z : 1)$ results in exchanging an infinite number of messages between them. For instance, for Block, introduced in Example 5, the infinite computation of $\cdot \Vdash y \leftarrow \mathtt{Loop} \mid_y z \leftarrow \mathtt{Block} \leftarrow y :: (z : 1)$ without communication along z can be depicted as follows:

$$\begin{array}{ll}
y \leftarrow \mathtt{Loop} \mid_y z \leftarrow \mathtt{Block} \leftarrow y & \mapsto \\
Ry.\mu_{\mathsf{nat}}; Ry.s; y \leftarrow \mathtt{Loop} \mid_y z \leftarrow \mathtt{Block} \leftarrow y & \mapsto \\
Ry.\mu_{\mathsf{nat}}; Ry.s; y \leftarrow \mathtt{Loop} \mid_y \mathbf{case}\, Ly\ (\mu_{\mathsf{nat}} \Rightarrow \mathbf{case}\, Ly\ \cdots) & \mapsto \\
Ry.s; y \leftarrow \mathtt{Loop} \mid_y \mathbf{case}\, Ly\ (s \Rightarrow z \leftarrow \mathtt{Block} \leftarrow y \mid z \Rightarrow \mathbf{wait}\, Ly; \mathbf{close}\, Rz) & \mapsto \\
y \leftarrow \mathtt{Loop} \mid_y z \leftarrow \mathtt{Block} \leftarrow y & \mapsto \\
\ldots &
\end{array}$$

In this example, the strong progress property of computation is violated. The configuration does not communicate to the left or right and a never ending series of internal communications takes place. This internal loop is a result of the infinite number of unfolding messages sent by Loop without any unfolding message with higher priority being received by it. In other words, it is the result of Loop not being valid.

In general, strong progress of $\mathtt{P}_1 \mid_x \mathtt{P}_2$ can be violated if either $\mathtt{P}_1$ or $\mathtt{P}_2$ are not valid. In the next section we will see an example of a program with strong progress property that seems to preserve this property after being composed with any other valid program. We will show that neither our local validity algorithm nor the FS guard condition identifies it as valid. In fact, Theorem 9 implies that no effective procedure, including our algorithm, can recognize a maximal set Ξ of programs with the strong progress property that is closed under composition. DeYoung and Pfenning showed that any Turing machine can be represented as a process in subsingleton logic with equirecursive fixed points [DP16, Pfe16]. Using this result and Theorem 9, we can easily reduce the halting problem to identifying closed programs $\mathcal{P} := \langle V, S \rangle$ with $\cdot \vdash S :: x : 1$ that satisfy strong progress. Note that a closed program satisfying strong progress is always in the maximal set Ξ.

13. A Negative Result

In this section we provide a straightforward example of a program with the strong progress property that our algorithm cannot identify as valid. Intuitively, this program seems to preserve the strong progress property after being composed by other valid programs. We show that this example does not satisfy the FS guard condition, either.

Example 18. Define the signature

$$\begin{aligned}
\Sigma_5 := \mathsf{ctr} &=^1_\nu \&\{inc : \mathsf{ctr},\ \ val : \mathsf{bin}\}, \\
\mathsf{bin} &=^2_\mu \oplus\{b0 : \mathsf{bin}, b1 : \mathsf{bin}, \$: 1\}
\end{aligned}$$

and program $\mathcal{P}_{11} = \langle\{\mathtt{Bit0Ctr}, \mathtt{Bit1Ctr}, \mathtt{Empty}\}, \mathtt{Empty}\rangle$, where

$$\begin{array}{l}
x : \mathsf{ctr} \vdash y \leftarrow \mathtt{Bit0Ctr} \leftarrow x :: (y : \mathsf{ctr}) \\
x : \mathsf{ctr} \vdash y \leftarrow \mathtt{Bit1Ctr} \leftarrow x :: (y : \mathsf{ctr}) \\
\cdot \vdash y \leftarrow \mathtt{Empty} :: (y : \mathsf{ctr})
\end{array}$$

with

$$\begin{array}{lll}
y^{\beta} \leftarrow \mathtt{Bit0Ctr} \leftarrow x^{\alpha} = & [0, 0, 0, 0] & \\
\quad \mathbf{case}\, Ry^{\beta}\ (v_{ctr} \Rightarrow & [-1, 0, 0, 0] & y_1^{\beta+1} < y_1^{\beta}, y_2^{\beta+1} = y_2^{\beta} \\
\quad\quad \mathbf{case}\, Ry^{\beta+1}\ (inc \Rightarrow\ y^{\beta+1} \leftarrow \mathtt{Bit1Ctr} \leftarrow x^{\alpha} & [-1, 0, 0, 0] & \\
\quad\quad\quad\quad \mid val \Rightarrow Ry^{\beta+1}.\mu_{bin}; Ry^{\beta+2}.b0; Lx^{\alpha}.v_{ctr}; Lx^{\alpha+1}.val; y^{\beta+2} \leftarrow x^{\alpha+1})) & [-1, 1, 0, 1] &
\end{array}$$

$$\begin{array}{lll}
y^{\beta} \leftarrow \mathtt{Bit1Ctr} \leftarrow x^{\alpha} = & [0, 0, 0, 0] & \\
\quad \mathbf{case}\, Ry^{\beta}\ (v_{ctr} \Rightarrow & [-1, 0, 0, 0] & y_1^{\beta+1} < y_1^{\beta}, y_2^{\beta+1} = y_2^{\beta} \\
\quad\quad \mathbf{case}\, Ry^{\beta+1}\ (inc \Rightarrow\ Lx^{\alpha}.v_{ctr}; Lx^{\alpha+1}.inc; y^{\beta+1} \leftarrow \mathtt{Bit0Ctr} \leftarrow x^{\alpha+1} & [-1, 1, 0, 0] & x_2^{\alpha+1} = x_2^{\alpha} \\
\quad\quad\quad\quad \mid val \Rightarrow Ry^{\beta+1}.\mu_{bin}; Ry^{\beta+2}.b1; Lx^{\alpha}.v_{ctr}; Lx^{\alpha+1}.val; y^{\beta+2} \leftarrow x^{\alpha+1})) & [-1, 1, 0, 1] &
\end{array}$$

$$\begin{array}{lll}
y^{\beta} \leftarrow \mathtt{Empty} \leftarrow \cdot = & [0, \_, \_, 0] & \\
\quad \mathbf{case}\, Ry^{\beta}\ (v_{ctr} \Rightarrow & [-1, \_, \_, 0] & y_1^{\beta+1} < y_1^{\beta}, y_2^{\beta+1} = y_2^{\beta} \\
\quad\quad \mathbf{case}\, Ry^{\beta+1}\ (inc \Rightarrow\ w^0 \leftarrow \mathtt{Empty} \leftarrow \cdot; & [\infty, \_, \_, \infty] & \mathsf{ctr}, \mathsf{bin} \in \mathsf{c}(\mathsf{ctr}) \\
\quad\quad\quad\quad\quad\quad y^{\beta+1} \leftarrow \mathtt{Bit1Ctr} \leftarrow w^0 & [-1, \infty, \infty, 0] & \mathsf{ctr}, \mathsf{bin} \in \mathsf{c}(ctr) \\
\quad\quad\quad\quad \mid val \Rightarrow Ry^{\beta+1}.\mu_{bin}; Ry^{\beta+2}.\$; \mathbf{close}\, Ry^{\beta+2})) & [-1, \_, \_, 1] &
\end{array}$$

In this example we implement a counter slightly differently from Example 15. We have two processes `Bit0Ctr` and `Bit1Ctr` that are holding one bit ($b0$ and $b1$ respectively) and an empty counter `Empty` that signals the end of the chain of counting processes. This program begins with an empty counter, if a value is requested, then it sends \$ to the right and if an increment is requested it adds the counter `Bit1Ctr` with $b1$ value to the chain of counters. Then if another increment is asked, `Bit1Ctr` sends an increment (inc) message to its left counter (implementing the carry bit) and calls `Bit0Ctr`. If `Bit0Ctr` receives an increment from the right, it calls `Bit1Ctr` recursively.

All (mutually) recursive calls in this program are recognized as valid by our algorithm, except the one in which `Empty` calls itself. In this recursive call, $y^{\beta} \leftarrow \mathtt{Empty} \leftarrow \cdot$ calls $w^0 \leftarrow \mathtt{Empty} \leftarrow \cdot$, where w is the fresh channel it shares with $y^{\beta+1} \leftarrow \mathtt{Bit1Ctr} \leftarrow w^0$. The number of increment unfolding messages `Bit1Ctr` can send along channel w^0 are always less than or equal to the number of increment unfolding messages it receives along channel $y^{\beta+1}$. This implies that the number of messages $w^0 \leftarrow \mathtt{Empty} \leftarrow \cdot$ may receive along channel w^0 is strictly less than the number of messages received by any process along channel y^{β}. There will be no infinite loop in the program without receiving an unfolding message from the right. Indeed Fortier and Santocanale's cut elimination for the cut corresponding to the composition `Empty` | `Bit1Ctr` locally terminates. Furthermore, since no valid program defined on the same signature can send infinitely many increment messages to the left, $\mathcal{P}_{11}$ composed with any other valid program satisfies strong progress.

This result is also a negative example for the FS guard condition. The path between $y^{\beta} \leftarrow \texttt{Empty} \leftarrow \cdot$ and $w^{0} \leftarrow \texttt{Empty} \leftarrow \cdot$ in the `Empty` process is neither left traceable not right traceable since $w \neq y$. By Definition 12 it is therefore not a valid cycle. △

14. Concluding Remarks

Related work. The main inspiration for this paper is work by Fortier and Santocanale [FS13], who provided a validity condition for pre-proofs in singleton logic with least and greatest fixed points. They showed that valid circular proofs in this system enjoy cut elimination. Also related is work by Baelde et al. [BDS16, Bae12], in which they similarly introduced a validity condition on the pre-proofs in multiplicative-additive linear logic with fixed points and proved the cut-elimination property for valid derivations. Doumane [Dou17] proved that this condition can be decided in *PSPACE* by reducing it to the emptiness problem of Büchi automata. Nollet et al. [NST18] introduced a local polynomial time algorithm for identifying a stricter version of Baelde's condition. At present, it is not clear to us how their algorithm would compare with ours on the subsingleton fragment due to the differences between classical and intuitionistic sequents [Lau18], different criteria on locality, and the prevailing computational interpretation of cut reduction as communicating processes in our work. Cyclic proofs have also been used for reasoning about imperative programs with recursive procedures [RB17]. While there are similarities (such as the use of cycles to capture recursion), their system extends separation logic is therefore not based on an interpretation of cut reduction as computation. Reasoning in their logic therefore has a very different character from ours. DeYoung and Pfenning [DP16] provide a computational interpretation of subsingleton logic with equirecursive fixed points and showed that cut reduction on circular pre-proofs in this system has the computational power of Turing machines. Their result implies undecidability of determining all programs with a strong progress property.

Our contribution. In this paper we have established an extension of the Curry-Howard interpretation for intuitionistic linear logic by Caires et al. [CP10, CPT16] to include least and greatest fixed points that can mutually depend on each other in arbitrary ways, although restricted to the subsingleton fragment. The key is to interpret circular pre-proofs in subsingleton logic as mutually recursive processes, and to develop a locally checkable, compositional validity condition on such processes. We proved that our local condition implies Fortier and Santocanale's guard condition and therefore also implies cut elimination. Analyzing this result in more detail leads to a computational strong progress property which means that a valid program will always terminate either in an empty configuration or one attempting to communicate along external channels.

Implementation. We have implemented the algorithm introduced in Section 10 in SML, which is publicly available [DDP19]. The implementation collects constraints and uses them to construct a suitable priority ordering over type variables and a $\subset$ ordering over process variables if they exist, and rejects the program otherwise. It also supports an implicit syntax where the fixed point unfolding messages are synthesized from the given communication patterns. Our experience with a range of programming examples shows that our local validity condition is surprisingly effective. Its main shortcoming arises when, intuitively, we need to know that a program's output is "smaller" than its input.

Future work. The main path for the future work is to extend our results to full ordered or linear logic with fixed points and address the known shortcomings we learned about through the implementation. We would also like to investigate the relationship to work by Nollet et al. [NST18], carried out in a different context.

Acknowledgments

The authors would like to acknowledge helpful comments by Stephanie Balzer, Ankush Das, Jan Hoffmann, and Siva Somayyajula on an earlier draft, discussions with Jonas Frey and Henry DeYoung regarding the subject of this paper, and support from the National Science Foundation under grant CCF-1718267 *Enriching Session Types for Practical Concurrent Programming*.

References

[Bae12] David Baelde. Least and greatest fixed points in linear logic. *ACM Transactions on Computational Logic*, 13(1):2:1–2:44, 2012.

[BDS16] David Baelde, Amina Doumane, and Alexis Saurin. Infinitary proof theory: the multiplicative additive case. In J.-M. Talbot and L. Regnier, editors, *25th Annual Conference on Computer Science Logic (CSL 2016)*, pages 42:1–42:17, Marseille, France, August 2016. LIPIcs 62.

[BM13] Lars Birkedal and Rasmus Ejlers Møgelberg. Intensional type theory with guarded recursive types qua fixed points on universes. In *28th Annual Symposium on Logic in Computer Science (LICS 2013)*, pages 213–222, New Orleans, LA, USA, June 2013. IEEE Computer Society.

[CCP03] Bor-Yuh Evan Chang, Kaustuv Chaudhuri, and Frank Pfenning. A judgmental analysis of linear logic. Technical Report CMU-CS-03-131R, Carnegie Mellon University, Department of Computer Science, December 2003.

[CP10] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010)*, pages 222–236, Paris, France, August 2010. Springer LNCS 6269.

[CPT16] Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 26(3):367–423, 2016. Special Issue on Behavioural Types.

[DCPT12] Henry DeYoung, Luís Caires, Frank Pfenning, and Bernardo Toninho. Cut reduction in linear logic as asynchronous session-typed communication. In P. Cégielski and A. Durand, editors, *Proceedings of the 21st Annual Conference on Computer Science Logic (CSL 2012)*, pages 228–242, Fontainebleau, France, September 2012. LIPIcs 16.

[DDP19] Ankush Das, Farzaneh Derakhshan, and Frank Pfenning. Subsingleton. `https://bitbucket.org/fpfenning/subsingleton`, June 2019. An implementation of subsingleton logic with ergometric and temporal types.

[DeY19] Henry DeYoung. *Session-Typed Ordered Logical Specifications*. PhD thesis, Computer Science Department, Carnegie Mellon University, 2019. Forthcoming.

[Dou17] Amina Doumane. *On the Infinitary Proof Theory of Logics with Fixed Points*. PhD thesis, Paris Diderot University, France, June 2017.

[DP16] Henry DeYoung and Frank Pfenning. Substructural proofs as automata. In A. Igarashi, editor, *14th Asian Symposium on Programming Languages and Systems*, pages 3–22, Hanoi, Vietnam, November 2016. Springer LNCS 10017. Invited talk.

[FS13] Jérôme Fortier and Luigi Santocanale. Cuts for circular proofs: Semantics and cut-elimination. In Simona Ronchi Della Rocca, editor, *22nd Annual Conference on Computer Science Logic (CSL 2013)*, pages 248–262, Torino, Italy, September 2013. LIPIcs 23.

[GL87] Jean-Yves Girard and Yves Lafont. Linear logic and lazy computation. In H. Ehrig, R. Kowalski, G. Levi, and U. Montanari, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, volume 2, pages 52–66, Pisa, Italy, March 1987. Springer-Verlag LNCS 250.

[Gra16] Hans Brugge Grathwohl. *Guarded Recursive Type Theory*. PhD thesis, Department of Computer Science, Aarhus University, Denmark, September 2016.

[Gri16] Dennis Griffith. *Polarized Substructural Session Types*. PhD thesis, University of Illinois at Urbana-Champaign, April 2016.

[GV10] Simon J. Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(1):19–50, January 2010.

[Hon93] Kohei Honda. Types for dyadic interaction. In *4th International Conference on Concurrency Theory*, CONCUR'93, pages 509–523. Springer LNCS 715, 1993.

[How69] W. A. Howard. The formulae-as-types notion of construction. Unpublished note. An annotated version appeared in: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, 479–490, Academic Press (1980), 1969.

[HVK98] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *7th European Symposium on Programming Languages and Systems (ESOP 1998)*, pages 122–138. Springer LNCS 1381, 1998.

[Lau18] Olivier Laurent. Around classical and intuitionistic linear logic. In A. Dawar and E. Grädel, editors, *Proceedings of the 33rd Annual Symposium on Logic in Computer Science (LICS 2018)*, pages 629–638, Oxford, UK, July 2018. ACM.

[LM16] Sam Lindley and J. Garrett Morris. Talking bananas: Structural recursion for session types. In J. Garrigue, G. Keller, and E. Sumii, editors, *Proceedings of the 21st International Conference on Functional Programming*, pages 434–447, Nara, Japan, September 2016. ACM Press.

[NST18] Rémi Nollet, Alexis Saurin, and Christine Tasson. Local validity for circular proofs in linear logic with fixed points. In D. Ghica and A. Jung, editors, *27th Annual Conference on Computer Science Logic (CSL 2018)*, pages 35:1–35:23. LIPIcs 119, 2018.

[Pfe16] Frank Pfenning. Substructural logics. Lecture notes for course given at Carnegie Mellon University, Fall 2016, December 2016.

[RB17] Reuben N. S. Rowe and James Brotherston. Automatic cyclic termination proofs for recursive procedures in separation logic. In Y. Bertot and V. Vafeiadis, editors, *Proceedings of the 6th Conference on Certified Programs and Proofs (CPP 2017)*, pages 53–65, Paris, France, January 2017. ACM.

[San02a] Luigi Santocanale. A calculus of circular proofs and its categorical semantics. In M. Nielsen and U. Engberg, editors, *5th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2002*, pages 357–371, Grenoble, France, April 2002. Springer LNCS 2303.

[San02b] Luigi Santocanale. μ-bicomplete categories and parity games. *Informatique Théorique et Applications*, 36(2):195–227, 2002.

[TCP13] Bernardo Toninho, Luís Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In M.Felleisen and P.Gardner, editors, *Proceedings of the European Symposium on Programming (ESOP'13)*, pages 350–369, Rome, Italy, March 2013. Springer LNCS 7792.

[Wad12] Philip Wadler. Propositions as sessions. In *Proceedings of the 17th International Conference on Functional Programming*, ICFP 2012, pages 273–286, Copenhagen, Denmark, September 2012. ACM Press.

Appendix A.

Here we provide the proof for the observations we made in Section 11. We prove that every program accepted by our algorithm in Section 10 corresponds to a valid circular proof in the sense of the FS guard condition. As explained in Section 6, the reflexive transitive closure of Ω in judgment $x^\gamma : \omega \vdash_\Omega P :: (y^\delta : C)$ forms a partial order $\leq_\Omega$. To enhance readability of proofs, throughout this section we use entailment $\Omega \Vdash x \leq y$ instead of $x \leq_\Omega y$.

We first prove Lemmas 10 and 11. Theorem 5 is a direct corollary of these two lemmas.

Lemma 10. Consider a path $\mathbb{P}$ on a program $Q = \langle V, S \rangle$ defined on a Signature Σ

$$\cfrac{x^\gamma : \omega' \vdash_{\Omega'} P' :: (y^\delta : C')}{\cfrac{\vdots}{z^\alpha : \omega \vdash_\Omega P :: (w^\beta : C)}}$$

with n the maximum priority in Σ.

(a) For every $i \in \mathsf{c}(\omega')$ with $\epsilon_i = \mu$, if $\Omega' \Vdash x^\gamma_i \leq z^\alpha_i$ then $x = z$ and $i \in \mathsf{c}(\omega)$.
(b) For every $i < n$, if $\Omega' \Vdash x^\gamma_i < z^\alpha_i$, then $i \in \mathsf{c}(\omega)$ and a μL rule with priority i is applied on $\mathbb{P}$.
(c) For every $c \leq n$ with $\epsilon_c = \nu$, if $\Omega' \Vdash x^\gamma_c \leq z^\alpha_c$, then no νL rule with priority c is applied on $\mathbb{P}$.

Proof. Proof is by induction on the structure of $\mathbb{P}$. We consider each case for last (topmost) step in $\mathbb{P}$.

Case:

$$\dfrac{x^\gamma : \omega' \vdash_{\Omega'} P' :: (y^\delta : C') \quad x : \omega' \vdash X = P' :: (y : C') \in V}{x^\gamma : \omega' \vdash_{\Omega'} y^\delta \leftarrow X \leftarrow x^\gamma :: (y^\delta : C')}\textsc{Def}(X)$$

None of the conditions in the conclusion are different from the premise. Therefore, by the induction hypothesis, statements (a)-(c) hold.

Case:

$$\dfrac{x^\gamma : \omega' \vdash_{\Omega'' \cup \mathtt{r}(v^\theta)} P'_{y^0} :: (y^0 : C') \quad y^0 : C' \vdash_{\Omega'' \cup \mathtt{r}(x^\gamma)} Q_{y^0} :: (v^\theta : C'')}{x^\gamma : \omega' \vdash_{\Omega''} (y \leftarrow P'_y; Q_y) :: (v^\theta : C'')}\textsc{Cut}^y$$

where $\mathtt{r}(u) = \{y^0_{p(s)} = u_{p(s)} \mid p(s) \notin \mathsf{c}(C')\}$ and $\Omega' = \Omega'' \cup r(v^\theta)$. All conditions in the conclusion are the same as the premise. Therefore, by the induction hypothesis, statements (a)-(c) hold.

Case:

$$\dfrac{u^\eta : \omega'' \vdash_{\Omega'' \cup \mathtt{r}(y^\delta)} Q_{a^0} :: (x^0 : A) \quad x^0 : A \vdash_{\Omega'' \cup \mathtt{r}(u^\eta)} P'_{x^0} :: (y^\delta : C'')}{u^\eta : \omega'' \vdash_{\Omega''} (x \leftarrow Q_x; P'_x) :: (y^\delta : C')}\textsc{Cut}^x$$

where $\mathtt{r}(v) = \{x^0_{p(s)} = v_{p(s)} \mid p(s) \notin \mathsf{c}(A)\}$ and $\Omega' = \Omega'' \cup r(u^\eta)$.

(a) $\Omega'' \cup r(u^\eta) \Vdash x^0_i \leq z^\alpha_i$ does not hold for any $i \in c(A)$ since x is a fresh channel. Therefore, this part is vacuously true.

(b) By freshness of x, if $\Omega'' \cup r(u^\eta) \Vdash x^0_i < z^\alpha_i$, then $x^0_i = u^\eta_i \in r(u^\eta)$ and $\Omega'' \Vdash u^\eta_i < z^\alpha_i$. By the induction hypothesis, $i \in \mathsf{c}(\omega)$ and a μL rule with priority i is applied on $\mathbb{P}$.

(c) By freshness of x, if $\Omega'' \cup r(u^\eta) \Vdash x^0_c \leq z^\alpha_c$, then $x^0_c = u^\eta_c \in r(u^\eta)$ and $\Omega'' \Vdash u^\eta_c \leq z^\alpha_c$. By the induction hypothesis, no νL rule with priority c is applied on $\mathbb{P}$.

Case:

$$\dfrac{. \vdash_{\Omega'} P' :: (y^\delta : C')}{u^\eta : 1 \vdash_{\Omega'} \textbf{wait}\, L u^\eta; P' :: (y^\delta : C')}1L$$

This case is not applicable since by the typing rules $\Omega' \nVdash . \leq z^\alpha_i$ for any $i \leq n$.

Case:

$$\dfrac{x^\alpha : \omega' \vdash_{\Omega'} P :: (y^{\delta'+1} : C') \quad t =_\mu C' \quad \Omega' = \Omega'' \cup \{(y^{\delta'})_{p(s)} = (y^{\delta'+1})_{p(s)} \mid p(s) \neq p(t)\}}{x^\alpha : \omega' \vdash_{\Omega''} R y^{\delta'}.\mu_t; P :: (y^{\delta'} : t)}\mu R$$

For every $i \leq n$, if $\Omega'' \cup \{(y^{\delta'})_{p(s)} = (y^{\delta'+1})_{p(s)} \mid p(s) \neq p(t)\} \Vdash x^\gamma_i \leq z^\alpha_i$, then $\Omega'' \Vdash x^\gamma_i \leq z^\alpha_i$. Therefore, by the induction hypothesis, statements (a)-(c) hold.

Case:

$$\dfrac{x^{\gamma'+1} : \omega' \vdash_{\Omega'} P' :: (y^\delta : C') \quad t =_\mu \omega' \quad \Omega' = \Omega'' \cup \{x^{\gamma'+1}_{p(t)} < x^{\gamma'}_{p(t)}\} \cup \{x^{\gamma'+1}_{p(s)} = x^{\gamma'}_{p(s)} \mid p(s) \neq p(t)\}}{x^{\gamma'} : t \vdash_{\Omega''} \textbf{case}\, L x^{\gamma'}\, (\mu_t \Rightarrow P') :: (y^\delta : C')}\mu L$$

By definition of $\mathsf{c}(x)$, we have $\mathsf{c}(\omega') \subseteq \mathsf{c}(t)$. By μL rule, for all $i \leq n$, $x^{\gamma'+1}_i \leq x^{\gamma'}_i \in \Omega'$. But by freshness of channels and their generations, $x^{\gamma'+1}$ is not involved in any relation in Ω''.

(a) For every $i \in \mathsf{c}(\omega')$ with $\epsilon_i = \mu$, if $\Omega' \Vdash x^{\gamma'+1}_i \leq z^\alpha_i$ then $\Omega'' \Vdash x^{\gamma'}_i \leq z^\alpha_i$. By the induction hypothesis, we have $x = z$ and $i \in \mathsf{c}(\omega)$.

(b) We consider two subcases: (1) If $\Omega' \Vdash x^{\gamma'+1}_i < z^\alpha_i$ for $i \neq p(t)$, then $\Omega' \Vdash x^{\gamma'+1}_i = x^{\gamma'}_i$ and $\Omega'' \Vdash x^{\gamma'}_i < z^\alpha_i$. Now we can apply the induction hypothesis. (2) If $\Omega' \Vdash x^{\gamma'+1}_{p(t)} < z^\alpha_{p(t)}$, then $\Omega' \Vdash x^{\gamma'+1}_{p(t)} < x^{\gamma'}_{p(t)}$ and $\Omega'' \Vdash x^{\gamma'}_{p(t)} \leq z^\alpha_{p(t)}$. Since a μL rule is applied in this step on the priority

$p(t)$, we only need to prove that $p(t) \in \mathsf{c}(\omega)$. By definition of c, we have $p(t) \in \mathsf{c}(t)$ and we can use the induction hypothesis on part (a) to get $p(t) \in \mathsf{c}(\omega)$.

(c) For every $c \leq n$ with $\epsilon_c = \nu$, $\Omega' \Vdash x_c^{\gamma'+1} = x_c^{\gamma'}$ as $c \neq p(t)$. Therefore, if $\Omega' \Vdash x_c^{\gamma'+1} = z_c^{\alpha}$, then $\Omega'' \Vdash x_c^{\gamma'} = z_c^{\alpha}$. By the induction hypothesis no νL rule with priority c is applied on $\mathbb{P}$.

Case:

$$\frac{x^{\gamma} : \omega' \vdash_{\Omega'} P' :: y^{\delta'+1} : C' \quad t =_{\nu} C' \quad \Omega' = \Omega'' \cup \{y_{p(t)}^{\delta'+1} < y_{p(t)}^{\delta'}\} \cup \{y_{p(s)}^{\delta'+1} = y_{p(s)}^{\delta'} \mid p(s) \neq p(t)\}}{x^{\gamma} : \omega' \vdash_{\Omega''} \mathbf{case}\, Ry^{\delta'}\ (\nu_t \Rightarrow P') :: (y^{\delta'} : t)}\nu R$$

For every $i \leq n$, if $\Omega'' \cup \{y_{p(t)}^{\delta'+1} < y_{p(t)}^{\delta'}\} \cup \{y_{p(s)}^{\delta'+1} = y_{p(s)}^{\delta'} \mid p(s) \neq p(t)\} \Vdash x_i^{\gamma} \leq z_i^{\alpha}$, then $\Omega'' \Vdash x_i^{\gamma} \leq z_i^{\alpha}$. Therefore, by the induction hypothesis, statements (a)-(c) hold.

Case:

$$\frac{x^{\gamma'+1} : \omega' \vdash_{\Omega'} Q :: (y^{\delta} : C') \quad t =_{\nu} \omega' \quad \Omega' = \Omega'' \cup \{x_{p(s)}^{\gamma'+1} = x_{p(s)}^{\gamma'} \mid p(s) \neq p(t)\}}{x^{\gamma'} : t \vdash_{\Omega''} Lx^{\gamma'}.\nu_t; P' :: (y^{\delta} : C')}\nu L$$

By definition of $\mathsf{c}(x)$, we have $\mathsf{c}(\omega') \subseteq \mathsf{c}(t)$. By νL rule, for all $i \neq p(t) \leq n$, $x_i^{\gamma'+1} = x_i^{\gamma'} \in \Omega'$. In particular, for every $i \leq n$ with $\epsilon_i = \mu$, $x_i^{\gamma'+1} = x_i^{\gamma'} \in \Omega'$. But by freshness of channels and their generations, $x^{\gamma'+1}$ is not involved in any relation in Ω''.

(a) For every $i \in \mathsf{c}(\omega')$ with $\epsilon_i = \mu$, if $\Omega' \Vdash x_i^{\gamma'+1} \leq z_i^{\alpha}$, then $\Omega' \Vdash x_i^{\gamma'+1} = x_i^{\gamma'}$ and $\Omega'' \Vdash x_i^{\gamma'} \leq z_i^{\alpha}$. By the induction hypothesis $x = z$ and $i \in \mathsf{c}(\omega)$.

(b) If $\Omega' \Vdash x_i^{\gamma'+1} < z_i^{\alpha}$, then by freshness of channels and their generations we have $i \neq p(t)$, $\Omega' \Vdash x_i^{\gamma'+1} = x_i^{\gamma'}$ and $\Omega'' \Vdash x_i^{\gamma'} < z_i^{\alpha}$. By the induction hypothesis $i \in \mathsf{c}(\omega)$ and a μL rule with priority i is applied on the path.

(c) For every $c \leq n$ with $\epsilon_c = \nu$ and $c \neq p(t)$, if $\Omega' \Vdash x_c^{\gamma'+1} \leq z_c^{\alpha}$, then $\Omega' \Vdash x_c^{\gamma'+1} = x_c^{\gamma'}$ and $\Omega'' \Vdash x^{\gamma'} \leq z_c^{\alpha}$. Therefore, by induction hypothesis, no νL rule with priority c is applied on the path. Note that $\Omega' \nVdash x_p^{\gamma'+1}(t) \leq z_p^{\alpha}(t)$.

Case:

$$\frac{x^{\gamma} : \omega \vdash_{\Omega'} Q_k :: y^{\delta} : A_k \quad \forall k \in L}{x^{\gamma} : \omega \vdash_{\Omega'} \mathbf{case}\, Ry^{\delta}\ (\ell \Rightarrow Q_\ell) :: (y^{\delta} : \&\{\ell : A_\ell\}_{\ell \in L})}\& R$$

None of the conditions in the conclusion are different from the premise. Therefore, by the induction hypothesis, statements (a)-(c) hold.

Case:

$$\frac{x^{\gamma} : A_k \vdash_{\Omega'} Q :: (y^{\delta} : C')}{x^{\gamma} : \&\{\ell : A_\ell\}_{\ell \in L} \vdash_{\Omega'} Lx^{\gamma}.k; Q :: (y^{\delta} : C')}\& L$$

By definition of $\mathsf{c}(x)$, we have $\mathsf{c}(A_k) \subseteq \mathsf{c}(\&\{\ell : A_\ell\}_{\ell \in L})$. Therefore, statements (a)-(c) follow from the induction hypothesis.

Cases: The statements are trivially true if the last step of the proof is either $1R$ or Id rules.

□

Lemma 11. Consider a path $\mathbb{P}$ on a program $Q = \langle V, S \rangle$ defined on a Signature Σ,

$$\dfrac{\dfrac{\overline{x}^{\gamma} : \omega' \vdash_{\Omega'} P' :: (y^{\delta} : C')}{\vdots}}{\overline{z}^{\alpha} : \omega \vdash_{\Omega} P :: (w^{\beta} : C)}$$

with n the maximum priority in Σ.

(a) For every $i \in \mathsf{c}(\omega')$ with $\epsilon_i = \nu$, if $\Omega' \Vdash y^{\delta}_i \leq w^{\beta}_i$, then $y = w$ and $i \in \mathsf{c}(\omega)$.
(b) If $\Omega' \Vdash y^{\delta}_i < w^{\beta}_i$, then $i \in \mathsf{c}(\omega)$ and a νL rule with priority i is applied on $\mathbb{P}$.
(c) For every $c \leq n$ with $\epsilon_c = \mu$, if $\Omega' \Vdash y^{\delta}_c \leq w^{\beta}_c$, then no μR rule with priority c is applied on $\mathbb{P}$.

Proof. Dual to the proof of Lemma 10. □

Lemma 12. Consider a path $\mathbb{P}$ on a program $Q = \langle V, S \rangle$ defined on a Signature Σ, with n the maximum priority in Σ.

$$\dfrac{\dfrac{\overline{x}^{\gamma} : \omega' \vdash_{\Omega'} P' :: (y^{\delta} : C')}{\vdots}}{\overline{z}^{\alpha} : \omega \vdash_{\Omega} P :: (w^{\beta} : C)}$$

Ω' preserves the (in)equalities in Ω. In other words, for channels u, v, generations $\eta, \eta' \in \mathbb{N}$ and type priorities $i, j \leq n$,

(a) If $\Omega \Vdash u^{\eta}_i < v^{\eta'}_j$, then $\Omega' \Vdash u^{\eta}_i < v^{\eta'}_j$.
(b) If $\Omega \Vdash u^{\eta}_i \leq v^{\eta'}_j$, then $\Omega' \Vdash u^{\eta}_i \leq v^{\eta'}_j$.
(c) If $\Omega \Vdash u^{\eta}_i = v^{\eta'}_j$, then $\Omega' \Vdash u^{\eta}_i = v^{\eta'}_j$.

Proof. Proof is by induction on the structure of $\mathbb{P}$. We consider each case for topmost step in $\mathbb{P}$. Here, we only give one non-trivial case. The proof of other cases is similar.

Case:

$$\dfrac{x^{\alpha} : \omega' \vdash_{\Omega'} P :: (y^{\delta'+1} : C') \quad t =_{\mu} C' \quad \Omega' = \Omega'' \cup \{(y^{\delta'})_{p(s)} = (y^{\delta'+1})_{p(s)} \mid p(s) \neq p(t)\}}{x^{\alpha} : \omega' \vdash_{\Omega''} Ry^{\delta'}.\mu_t; P :: (y^{\delta'} : t)}\mu R$$

(a) If $\Omega \Vdash u^{\eta}_i < v^{\eta'}_j$, then by the inductive hypothesis, $\Omega'' \Vdash u^{\eta}_i < v^{\eta'}_j$. By freshness of channels and their generations, we know that $y^{\delta'+1}$ does not occur in any (in)equalities in Ω'' and thus $y^{\delta'+1} \neq u^{\eta}, v^{\eta'}$. Therefore $\Omega' \Vdash u^{\eta}_i < v^{\eta'}_j$.

Following the same reasoning, we can prove statements (b) and (c).

□

Lemma 13. Consider a finitary derivation (Figure 4) for

$$\langle \overline{u}, X, v \rangle; \overline{x}^{\alpha} : \omega \vdash_{\Omega} P :: (y^{\beta} : C),$$

on a locally valid program $Q = \langle V, S \rangle$ defined on signature Σ and order $\subset$. There is a (potentially infinite) derivation $\mathbb{D}$ for

$$\overline{x}^{\alpha} : \omega \vdash_{\Omega} P :: (y^{\beta} : C),$$

based in the infinitary rule system of Figure 3.

Moreover, for every $\bar{w}^\gamma : \omega' \vdash_{\Omega'} z^\delta \leftarrow Y \leftarrow \bar{w}^\gamma :: (z^\delta : C')$ on $\mathbb{D}$, we have

$$Y, list(\bar{w}^\gamma, z^\delta) \; (\subset, <_{\Omega'}) \; X, list(\bar{x}^\alpha, y^\beta).$$

Proof. We prove this by coinduction, producing the derivation of $\bar{x}^\alpha : \omega \vdash_\Omega P :: (y^\beta : C)$. We proceed by case analysis of the first rule applied on $\langle \bar{u}, X, v \rangle; \bar{x}^\alpha : \omega \vdash_\Omega P :: (y^\beta : C)$, in its finite derivation.

Case:

$$\dfrac{Y, list(\bar{x}^\alpha, y^\beta) \; (\subset, <_\Omega) \; X, list(\bar{u}^\gamma, v^\delta) \quad \bar{x} : \omega \vdash Y = P'_{\bar{x},y} :: (y : C) \in V}{\langle \bar{u}^\gamma, X, v^\delta \rangle; \bar{x}^\alpha : \omega \vdash_{\Omega,\subset} y^\beta \leftarrow Y \leftarrow \bar{x}^\alpha :: (y^\beta : C)} \textsc{Call}(Y)$$

By validity of program there is a finitary derivation for

$$\langle \bar{x}^0; Y; y^0 \rangle; \bar{x}^0 : \omega \vdash_{\emptyset,\subset} P'_{\bar{x}^0, y^0} :: (y^0) : C.$$

Having Proposition 4 and freshness of future generations of channels in Ω, there is also a finitary derivation for

$$\langle \bar{x}^\alpha; Y; y^\beta \rangle; \bar{x}^\alpha : \omega \vdash_{\Omega,\subset} P'_{\bar{x}^\alpha, y^\beta} :: (y^\beta : C).$$

We apply the coinductive hypothesis to get an infinitary derivation $\mathbb{D}'$ for

$$\bar{x}^\alpha : \omega \vdash_{\Omega,\subset} P'_{\bar{x}^\alpha, y^\beta} :: (y^\beta : C),$$

and then produce the last step of derivation

$$\dfrac{\overset{\mathbb{D}'}{\bar{x}^\alpha : \omega \vdash_\Omega P'_{\bar{x}^\alpha, y^\beta} :: (y^\beta : C)} \quad \bar{x} : \omega \vdash Y = P'_{\bar{x},y} :: (y : C) \in V}{\bar{x}^\alpha : \omega \vdash_\Omega y^\beta \leftarrow Y \leftarrow \bar{x}^\alpha :: (y^\beta : C)} \textsc{Def}(Y)$$

in the infinitary rule system.

Moreover, by the coinductive hypothesis, we know that for every

$$\bar{w}^{\gamma'} : \omega' \vdash_{\Omega'} z^{\delta'} \leftarrow W \leftarrow w^{\gamma'} :: (z^{\delta'} : C')$$

on $\mathbb{D}'$, we have

$$W, list(w^{\gamma'}, z^{\delta'}) \; (\subset, <_{\Omega'}) \; Y, list(\bar{x}^\alpha, y^\beta).$$

By Lemma 12, we conclude from $Y, list(\bar{x}^\alpha, y^\beta) \; (\subset, <_\Omega) \; X, list(\bar{u}^\gamma, v^\delta)$ that

$$Y, list(\bar{x}^\alpha, y^\beta) \; (\subset, <_{\Omega'}) \; X, list(\bar{u}^\gamma, v^\delta).$$

By transitivity of $(\subset, <_{\Omega'})$, we get

$$W, list(w^{\gamma'}, z^{\delta'}) \; (\subset, <_{\Omega'}) \; X, list(\bar{u}^\gamma, v^\delta).$$

This completes the proof of this case as we already know $Y, list(\bar{x}^\alpha, y^\beta) \; (\subset, <_\Omega) \; X, list(\bar{u}^\gamma, v^\delta)$.

Case:

$$\dfrac{\langle \bar{u}^\gamma, X, v^\delta \rangle; \bar{x}^\alpha : \omega \vdash_{\Omega \cup \mathtt{r}(y^\beta),\subset} Q_{z^0} :: (z^0 : C') \quad \langle \bar{u}^\gamma, X, v^\delta \rangle; z^0 : C' \vdash_{\Omega \cup \mathtt{r}(x^\alpha),\subset} Q'_{z^0} :: (y^\beta : C)}{\langle \bar{u}^\gamma, X, v^\delta \rangle; \bar{x}^\alpha : \omega \vdash_{\Omega,\subset} (z \leftarrow Q_z; Q'_z) :: (y^\beta : C)} \textsc{Cut}^z,$$

where $\mathtt{r}(w) = \{z^0_{p(s)} = w_{p(s)} \mid p(s) \notin \mathtt{c}(A)\}$. By coinductive hypothesis, we have infinitary derivations $\mathbb{D}'$ ad $\mathbb{D}''$ for $\bar{x}^\alpha : \omega \vdash_{\Omega \cup \mathtt{r}(y^\beta)} Q_{z^0} :: (z^0 : C')$ and $z^0 : C' \vdash_{\Omega \cup \mathtt{r}(x^\alpha)} Q'_{z^0} :: (y^\beta : C)$, respectively. We can produce the last step of the derivation as

$$\dfrac{\begin{matrix}\mathbb{D}' \\ \bar{x}^\alpha : \omega \vdash_{\Omega \cup \mathtt{r}(y^\beta)} Q_{z^0} :: (z^0 : C')\end{matrix} \quad \begin{matrix}\mathbb{D}'' \\ z^0 : C' \vdash_{\Omega \cup \mathtt{r}(x^\alpha)} Q'_{z^0} :: (y^\beta : C)\end{matrix}}{\bar{x}^\alpha : \omega \vdash_\Omega (z \leftarrow Q_z ; Q'_z) :: (y^\beta : C)}\textsc{Cut}^z$$

Moreover, by the coinductive hypothesis, we know that for every

$$\bar{w}^\gamma : \omega' \vdash_{\Omega'} z^\delta \leftarrow W \leftarrow \bar{w}^\gamma :: (z^\delta : C')$$

on $\mathbb{D}'$ and $\mathbb{D}''$, and thus $\mathbb{D}$, we have

$$W, list(\bar{w}^\gamma, z^\delta)\ (\subset, <_{\Omega'})\ X, list(\bar{x}^\alpha, y^\beta).$$

Cases: The proof of the other cases are similar by applying the coinductive hypothesis and the infinitary system rules.

□

Theorem 6. A locally valid program satisfies the FS guard condition.

Proof. Consider a cycle $\mathbb{C}$ on a (potentially infinite) derivation produced from $\langle \bar{u}, Y, v\rangle ; \bar{z}^\alpha : \omega \vdash_\Omega w^\beta \leftarrow X \leftarrow \bar{z}^\alpha :: (w^\beta : C)$ as in Lemma 13,

$$\dfrac{\bar{x}^\gamma : \omega \vdash_{\Omega'} P_{x^\gamma, y^\delta} :: (y^\delta : C) \quad \bar{z} : \omega \vdash X = P_{\bar{z}, w} :: (w : C) \in V}{\underline{\bar{x}^\gamma : \omega \vdash_{\Omega'} y^\delta \leftarrow X \leftarrow \bar{x}^\gamma :: (y^\delta : C)}}\textsc{Def}(X)$$

$$\vdots$$

$$\dfrac{\overline{\bar{z}^\alpha : \omega \vdash_\Omega P_{z^\alpha, w^\beta} :: (w^\beta : C)} \qquad \bar{z} : \omega \vdash_X = P_{\bar{z}, w} :: (w : C) \in V}{\bar{z}^\alpha : \omega \vdash_\Omega w^\beta \leftarrow X \leftarrow \bar{z}^\alpha :: (w^\beta : C)}\textsc{Def}(X)$$

By Lemma 13 we get

$$X, list(\bar{x}^\gamma, y^\delta)\ (\subset, <_{\Omega'})\ X, list(\bar{z}^\alpha, w^\beta),$$

and thus by definition of $(\subset, <_{\Omega'})$,

$$list(\bar{x}^\gamma, y^\delta) <_{\Omega'} list(\bar{z}^\alpha, w^\beta).$$

Therefore, there is an $i \leq n$, such that either

(1) $\epsilon_i = \mu$, $x_i^\gamma < z_i^\alpha$, and $x_l^\gamma = z_l^\alpha$ for every $l < i$, having that $\bar{x} = x$ and $\bar{z} = z$ are non-empty, or

(2) $\epsilon_i = \nu$, $y_i^\delta < w_i^\beta$, and $y_l^\delta = w_l^\beta$ for every $l < i$.

In the first case, by part (b) of Lemma 10, a μL rule with priority $i \in \mathsf{c}(\omega)$ is applied on $\mathbb{C}$. By part (a) of the same Lemma $x = z$, and by its part (c), no νL rule with priority $c < i$ is applied on $\mathbb{C}$. Therefore, $\mathbb{C}$ is a left μ- trace.

In the second case, by part (b) of Lemma 11, a νR rule with priority $i \in \mathsf{c}(\omega)$ is applied on $\mathbb{C}$. By part (a) of the same Lemma $y = w$ and by its part (c), no μR rule with priority $c < i$ is applied on $\mathbb{C}$. Thus, $\mathbb{C}$ is a right ν- trace. □