Nested Session Types

ANKUSH DAS and HENRY DEYOUNG, Carnegie Mellon University
ANDREIA MORDIDO, Universidade de Lisboa
FRANK PFENNING, Carnegie Mellon University

Session types statically describe communication protocols between concurrent message-passing processes. Unfortunately, parametric polymorphism even in its restricted prenex form is not fully understood in the context of session types. In this article, we present the metatheory of session types extended with prenex polymorphism and, as a result, nested recursive datatypes. Remarkably, we prove that type equality is decidable by exhibiting a reduction to trace equivalence of deterministic first-order grammars. Recognizing the high theoretical complexity of the latter, we also propose a novel type equality algorithm and prove its soundness. We observe that the algorithm is surprisingly efficient and, despite its incompleteness, sufficient for all our examples. We have implemented our ideas by extending the Rast programming language with nested session types. We conclude with several examples illustrating the expressivity of our enhanced type system.

CCS Concepts: • **Software and its engineering** → **Polymorphism**; **Recursion**; *Concurrent programming languages*; **Functional languages;**

Additional Key Words and Phrases: Nested types, polymorphism, type equality

ACM Reference format:
Ankush Das, Henry DeYoung, Andreia Mordido, and Frank Pfenning. 2022. Nested Session Types. *ACM Trans. Program. Lang. Syst.* 44, 3, Article 19 (July 2022), 45 pages.
https://doi.org/10.1145/3539656

1 INTRODUCTION

Session types express and enforce interaction protocols in message-passing systems [32, 50]. In this work, we focus on *binary session types* that describe bilateral protocols between two endpoint processes performing dual actions. Binary session types obtained a firm logical foundation since they were shown to be in a Curry-Howard correspondence with linear logic propositions [7, 8, 53]. This allows us to rely on properties of cut reduction to derive type safety properties such as *progress (deadlock freedom)* and *preservation (session fidelity)*, which continue to hold even when extended to recursive types and processes [18].

Support for this research was provided by the Fundação para a Ciência e a Tecnologia through the Carnegie Mellon Portugal Program—Visiting Faculty Program, the project SafeSessions (PTDC/CCI-COM/6453/2020), and the LASIGE Research Unit (UIDB/00408/2020 and UIDP/00408/2020), and by the National Science Foundation under SaTC Award 1801369, CAREER Award 1845514 and Grant No. 1718276.

Authors' addresses: A. Das, Amazon, 20450 Stevens Creek Blvd, Cupertino, CA 95014; email: daankus@amazon.com; H. DeYoung and F. Pfenning, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213; emails: {hdeyoung, fp}@cs.cmu.edu; A. Mordido, LASIGE, Faculdade de Ciências, Universidade de Lisboa, Lisboa, Portugal; email: afmordido@ciencias.ulisboa.pt.

0164-0925/2022/07-ART19
https://doi.org/10.1145/3539656

However, the theory of session types is still missing a crucial piece: a general understanding of prenex (or ML-style) parametric polymorphism, encompassing recursively defined types, polymorphic type constructors, and nested types. We abbreviate the sum of these features simply as *nested types* [3]. Prior work has restricted itself to parametric polymorphism either in prenex form without nested types [28, 51], with explicit higher-rank quantifiers [6, 42] (including bounded ones [26]) but without general recursion, or in specialized form for iteration at the type level [52]. None of these allow a free, *nested* use of polymorphic type constructors combined with prenex polymorphism.

In this article, we develop the metatheory of this rich language of nested session types. Nested types are reasonably well understood in the context of functional languages [3, 34] and have several interesting applications [10, 31, 41]. One difficult point is the interaction of nested types with polymorphic recursion and type inference [40]. By adopting bidirectional type checking, we avoid this particular set of problems altogether, at the cost of some additional verbosity. However, we have a new problem, namely how to handle type equality ($\equiv$) given that session type definitions are generally *equirecursive* and *not generative*.

Consider the following list types, $\mathsf{list}[\alpha]$ and $\mathsf{list}'[\alpha]$, where $\oplus$ is the constructor for an *internal choice* between **nil** and **cons** labels, and where $\otimes$ is the constructor for a product type and $\mathbf{1}$ is its unit. Essentially, these types are the same as the usual functional datatype for lists.

$$\mathsf{list}[\alpha] \triangleq \oplus\{\mathbf{nil} : \mathbf{1}, \mathbf{cons} : \alpha \otimes \mathsf{list}[\alpha]\} \qquad \mathsf{list}'[\alpha] \triangleq \oplus\{\mathbf{nil} : \mathbf{1}, \mathbf{cons} : \alpha \otimes \mathsf{list}'[\alpha]\}$$

Even before we consider nesting, from these types alone we have $\mathsf{list}[A] \equiv \mathsf{list}'[B]$ and also $\mathsf{list}[\mathsf{list}'[A]] \equiv \mathsf{list}'[\mathsf{list}[B]]$, provided $A \equiv B$. The reason is that both types specify the same communication behavior—only their name (which is irrelevant in an equirecursive, non-generative setting) is different. As the second of these equalities shows, deciding the equality of nested occurrences of type constructors is inescapable: as soon as we allow type constructors (which are necessary in many practical examples) in combination with an equirecursive, non-generative interpretation of types, we also must solve type equality for nested types! In addition, as described in Section 6, type equality pervades type checking.

As another example in which nested types are used, the types $\mathsf{Tree}[\alpha]$ and $\mathsf{STree}[\alpha, \kappa]$ represent binary trees and their faithfully (and efficiently) serialized form, respectively.

$$\mathsf{Tree}[\alpha] \triangleq \oplus\{\mathbf{node} : \mathsf{Tree}[\alpha] \otimes \alpha \otimes \mathsf{Tree}[\alpha], \mathbf{leaf} : \mathbf{1}\}$$

$$\mathsf{STree}[\alpha, \kappa] \triangleq \oplus\{\mathbf{nd} : \mathsf{STree}[\alpha, \alpha \otimes \mathsf{STree}[\alpha, \kappa]], \mathbf{lf} : \kappa\}$$

Indeed, $\mathsf{STree}[\alpha, \kappa]$ describes serialized trees because it is isomorphic (but certainly not equal) to $\mathsf{Tree}[\alpha] \otimes \kappa$. In implementing the processes that serialize and deserialize trees (which happen to witness the type isomorphism), we make use of type equality for the nested type $\mathsf{STree}[\alpha, \kappa]$ (see Section 9). There is no one particular feature of this example that necessitates type equality—instead, type equality is used in type checking because of the interaction of type constructors with the equirecursive, non-generative interpretation of types.

At the core of type checking lies *type equality*. We show that we can translate type equality for nested session types to the trace equivalence problem for deterministic first-order grammars, shown to be decidable by Jančar [33], albeit with doubly-exponential complexity. Solomon [48] already proved a related connection between *inductive* type equality for nested types and language equality for **deterministic pushdown automata (DPDAs)**. The difference is that the standard session type equality is defined coinductively, as a bisimulation, rather than via language equivalence [25]. This is because session types capture communication behavior rather than the structure of closed values so a type such as $\mathsf{R} \triangleq \oplus\{\mathbf{a} : \mathsf{R}\}$ is not equal to the empty type $\mathsf{E} \triangleq \oplus\{\}$. The reason is that the former type can send infinitely many **a**'s while the latter cannot, and hence their communication behavior is different, implying that the types must be different. Interestingly, if we imagine a lazy functional language such as Haskell with non-generative recursive types, then R and E would also be different. In fact, nothing in our analysis of equirecursive nested types depends on linearity, just on the coinductive interpretation of types. Our key results, namely decidability of type equality and a practical algorithm for it, apply to lazy functional languages!

The decision procedure for deterministic first-order grammars does not appear to be directly suitable for implementation, in part due to its doubly-exponential complexity bound. Instead, we develop an algorithm combining loop detection [25] with instantiation [19] and a special treatment of reflexivity. The algorithm is sound but incomplete, and reports success, a counterexample, or an inconclusive outcome (which counts as failure). In our experience, the algorithm is surprisingly efficient and sufficient for all our examples.

We have implemented nested session types and integrated them with the Rast language that is based on session types [18–20]. We have evaluated our prototype on several examples such as the Dyck language [22], an expression server [51], serializing binary trees, and standard polymorphic data structures such as lists, stacks, and queues.

Most closely related to our work is **context-free session types (CFSTs)** [51]. CFSTs also enhance the expressive power of binary session types by extending types with a notion of sequential composition of types. In connection with CFSTs, we identified a proper fragment of nested session types closed under sequential composition, and therefore nested session types are strictly more expressive than CFSTs.

The main technical contributions of our work are as follows:

- A uniform language of session types supporting prenex polymorphism, type constructors, and nested types and its type safety proof (Sections 3 and 6).
- A proof of decidability of type equality (Section 4).
- A practical algorithm for type equality and its soundness proof (Section 5).
- A proper fragment of nested session types that is closed under sequential composition, the main feature of CFSTs (Section 7).
- An implementation and integration with the Rast language (Section 8).

This article is an extended version of conference paper at ESOP 2021 [14]. The major additions in this article include full proofs for our results (Sections 4 and 5), more examples illustrating the expressivity of our types and language (Section 9), and a more in-depth description of both the programming language (Section 6) and the type equality algorithm with its optimizations (Sections 4 and 5).

2 OVERVIEW OF NESTED SESSION TYPES

The main motivation for studying nested types is quite practical and generally applicable to programming languages with structural type systems. We start by applying parametric type constructors for a standard polymorphic queue data structure. We also demonstrate how the types can be made more precise using nesting. A natural consequence of having nested types is the ability to capture (communication) patterns characterized by context-free languages. As an illustration, we express the Dyck language of balanced parentheses and also show how nested types are connected to DPDAs.

2.1 Queues

A standard application of parameterized types is the definition of polymorphic data structures such as lists, stacks, or queues. As a simple example, consider the following polymorphic type.

$$\mathrm{Queue}[\alpha] \triangleq \&\{\mathbf{enq} : \alpha \multimap \mathrm{Queue}[\alpha], \mathbf{deq} : \oplus\{\mathbf{none} : \mathbf{1}, \mathbf{some} : \alpha \otimes \mathrm{Queue}[\alpha]\}\}$$

The type Queue, parameterized by α, represents a queue with values of type α. A process providing this type offers an *external choice* ($\&$) enabling the client to either *enqueue* a value of type α in the queue (label **enq**), or to *dequeue* a value from the queue (label **deq**). Upon receiving label **enq**, the provider expects to receive a value of type α (the $\multimap$ operator) and then proceeds to offer Queue$[\alpha]$. Upon receiving the label **deq**, the provider queue is either empty, in which case it sends the label **none** and terminates the session (as prescribed by type **1**), or is non-empty, in which case it sends a value of type α (the $\otimes$ operator) and recurses with Queue$[\alpha]$.

Although parameterized type definitions like the preceding one are sufficient to express the standard interfaces to polymorphic data structures, we propose *nested session types*, which are considerably more expressive. For instance, we can use type parameters to track the number of elements in the queue as part of its type!

$$\mathrm{Queue}[\alpha,\kappa] \triangleq \&\{\mathbf{enq} : \alpha \multimap \mathrm{Queue}[\alpha, \mathrm{Some}[\alpha, \mathrm{Queue}[\alpha,\kappa]]], \mathbf{deq} : \kappa\}$$

$$\mathrm{Some}[\alpha,\kappa] \triangleq \oplus\{\mathbf{some} : \alpha \otimes \kappa\} \qquad\qquad \mathrm{None} \triangleq \oplus\{\mathbf{none} : \mathbf{1}\}$$

The second type parameter, κ, tracks the number of elements. This parameter can be understood as a *symbol stack* in a DPDA. On enqueuing an element, we recurse to $\mathrm{Queue}[\alpha, \mathrm{Some}[\alpha, \mathrm{Queue}[\alpha,\kappa]]]$ denoting the *push* of the Some symbol onto stack κ. We initiate the empty queue with the type $\mathrm{Queue}[\alpha, \mathrm{None}]$ where the second parameter denotes an *empty symbol stack*. A queue with n elements has the type $\mathrm{Queue}[\alpha, \mathrm{Some}[\alpha, \mathrm{Queue}[\alpha, \ldots, \mathrm{Some}[\alpha, \mathrm{Queue}[\alpha, \mathrm{None}]] \cdots]]]$, where Some appears n times. On receipt of the deq label, the type transitions to κ, which can either be $\kappa = \mathrm{None}$ (if the queue is empty) or $\kappa = \mathrm{Some}[\alpha, \mathrm{Queue}[\alpha, \kappa']]$ (if the queue is non-empty). In the latter case, the type sends label **some** followed by an element and transitions to $\mathrm{Queue}[\alpha, \kappa']$ denoting a *pop* from the symbol stack. In the former case, the type sends the label none and terminates. Both of these behaviors are reflected in the definitions of the types Some and None. In Section 9, we elaborate the queue example by providing process definitions for empty and non-empty queues.

Alternatively, this example could be expressed with arithmetic type refinements (e.g., see [19]). However, with refinement types (and general dependent types), type equality can be undecidable even when the refinement layer is decidable (e.g., Presburger arithmetic [19]). With the nested session types presented in this article, type equality and type checking are decidable.

2.2 Context-Free Languages

Recursive session types capture the class of regular languages [51]. However, in practice, many useful languages are beyond regular. As an illustration, suppose we would like to express a balanced parentheses language, also known as the Dyck language [22] with the end-marker \$. We use **L** to denote an opening symbol and **R** to denote a closing symbol. (In a session-typed mindset, **L** can represent a client's request and **R** is the server's response). We need to enforce that each **L** has a corresponding closing **R** and they are properly nested. To express this, we need to track the number of **L**'s in the output with the session type. However, this notion of *memory* is beyond the expressive power of regular languages, so mere recursive session types will not suffice.

We utilize the expressive power of nested types to express this behavior.

$$D[\kappa] \triangleq \oplus\{\mathbf{L} : D[D[\kappa]], \mathbf{R} : \kappa\} \qquad D_0 \triangleq \oplus\{\mathbf{L} : D[D_0], \$: \mathbf{1}\}$$

The nested type $D[\kappa]$ takes κ as a type parameter and either outputs **L** and continues with $D[D[\kappa]]$ or outputs **R** and continues with κ. The type D_0 either outputs **L** and continues with $D[D_0]$ or outputs \$ and terminates. The type D_0 expresses a Dyck word with end-marker \$ [38].

The key idea here is that the number of D's in the type of a word tracks the number of unmatched $\mathbf{L}$'s in it. Whenever the type $D[\kappa]$ outputs $\mathbf{L}$, it recurses with $D[D[\kappa]]$ incrementing the number of D's in the type by 1. Dually, whenever the type outputs $\mathbf{R}$, it recurses with κ decrementing the number of Ds in the type by 1. The type D_0 denotes a balanced word with no unmatched $\mathbf{L}$'s. Moreover, since we can only output \$ (or $\mathbf{L}$) but *not* $\mathbf{R}$ at the type D_0, we obtain the invariant that any word of type D_0 must be balanced. If we imagine the parameter κ as the symbol stack in a DPDA, outputting an $\mathbf{L}$ pushes D on the stack, whereas outputting $\mathbf{R}$ pops D from the stack. The definition of D_0 ensures that once an $\mathbf{L}$ is output, the symbol stack is initialized with $D[D_0]$ indicating one unmatched $\mathbf{L}$.

Nested session types *do not* restrict communication to *balanced* represented words. Indeed, the type D'_0 can model the *cropped Dyck language*, the language of all prefixes of Dyck words (with end-marker \$). Stated differently, D'_0 describes those words that could eventually become well-balanced Dyck words (as described by D_0) if enough $\mathbf{R}$s were tacked on the end (before the terminal \$).

$$D'[\kappa] \triangleq \oplus\{\mathbf{L} : D'[D'[\kappa]], \mathbf{R} : \kappa, \$: \mathbf{1}\} \qquad\qquad D'_0 \triangleq \oplus\{\mathbf{L} : D'[D'_0], \$: \mathbf{1}\}$$

The only difference between types $D[\kappa]$ and $D'[\kappa]$ is that $D'[\kappa]$ allows us to terminate at any point using the \$ label which immediately transitions to type $\mathbf{1}$, and the only difference between types D_0 and D'_0 is that D'_0 uses $D'[\kappa]$ instead of $D[\kappa]$. Nested session types not only capture the class of deterministic context-free languages recognized by DPDAs that *accept by empty stack* (balanced words) but also the class of deterministic context-free languages recognized by DPDAs that *accept by final state* (cropped words).

2.2.1 Multiple Stack Symbols. Thus far, we have alluded to a relationship between nested types and DPDAs in which the nested type parameter acts like the symbol stack in a DPDA. Nested types are expressive enough to allow any finite alphabet of stack symbols to be used, not just the single-symbol stack alphabets implied by the preceding examples. As an example, consider the language of Dyck words over several kinds of parentheses. Let $\mathbf{L}$ and $\mathbf{L}'$ denote two kinds of opening symbols, whereas $\mathbf{R}$ and $\mathbf{R}'$ denote their corresponding closing symbols, respectively. We define the following session types.

$$\begin{array}{rcl} S[\kappa] & \triangleq & \oplus\{\mathbf{L} : S[S[\kappa]], \mathbf{L}' : S'[S[\kappa]], \mathbf{R} : \kappa\} \\ S'[\kappa] & \triangleq & \oplus\{\mathbf{L} : S[S'[\kappa]], \mathbf{L}' : S'[S'[\kappa]], \mathbf{R}' : \kappa\} \\ E & \triangleq & \oplus\{\mathbf{L} : S[E], \mathbf{L}' : S'[E], \$: \mathbf{1}\} \end{array}$$

We *push* symbols S and S' onto the stack when outputting $\mathbf{L}$ and $\mathbf{L}'$, respectively. Symmetrically, we *pop* S and S' from the stack when outputting $\mathbf{R}$ and $\mathbf{R}'$, respectively. Then, the type E defines an *empty stack*, thereby representing a balanced Dyck word.

From the perspective of session-typed concurrency, nested types can neatly capture *complex server-client interactions*. For instance, client requests can be captured using labels $\mathbf{L}, \mathbf{L}'$, whereas server responses can be captured using labels $\mathbf{R}, \mathbf{R}'$ expressing *multiple kinds* of requests. Balanced words will then represent that all requests have been handled. The types can also guarantee that the number of responses does not exceed the number of requests.

2.2.2 Multiple States as Multiple Parameters. Using defined type names with *multiple* type parameters, we enable types to capture the language of DPDAs with several states. Consider the language $L_3 = \{\mathbf{L}^n\,\mathbf{a}\,\mathbf{R}^n\,\mathbf{a} \cup \mathbf{L}^n\,\mathbf{b}\,\mathbf{R}^n\,\mathbf{b} \mid n > 0\}$, proposed by Korenjak and Hopcroft [38]. A word in this language starts with a sequence of opening symbols $\mathbf{L}$, followed by an *intermediate symbol*, either $\mathbf{a}$ or $\mathbf{b}$. Then, the word contains as many closing symbols $\mathbf{R}$ as there were $\mathbf{L}$s and terminates with the symbol **a** or **b** *matching* the intermediate symbol.

$$U \triangleq \oplus\{\mathbf{L} : O[C[A], C[B]]\} \qquad O[\kappa_{\mathbf{a}}, \kappa_{\mathbf{b}}] \triangleq \oplus\{\mathbf{L} : O[C[\kappa_{\mathbf{a}}], C[\kappa_{\mathbf{b}}]], \mathbf{a} : \kappa_{\mathbf{a}}, \mathbf{b} : \kappa_{\mathbf{b}}\}$$

$$C[\kappa] \triangleq \oplus\{\mathbf{R} : \kappa\} \qquad A \triangleq \oplus\{\mathbf{a} : \mathbf{1}\} \qquad B \triangleq \oplus\{\mathbf{b} : \mathbf{1}\}$$

The L_3 language is characterized by session type U. Since the type U is unaware of which intermediate symbol among **a** or **b** would eventually be chosen, it cleverly maintains *two symbol stacks* in the two type parameters $\kappa_{\mathbf{a}}$ and $\kappa_{\mathbf{b}}$ of O. We initiate type U with outputting **L** and transitioning to $O[C[A], C[B]]$, where the symbol C tracks that we have outputted *one* **L**. The types A and B represent the intermediate symbols that might be used in the future. The type $O[\kappa_{\mathbf{a}}, \kappa_{\mathbf{b}}]$ can either output an **L** and transition to $O[C[\kappa_{\mathbf{a}}], C[\kappa_{\mathbf{b}}]]$ *pushing* the symbol C onto *both* stacks or can output **a** (or **b**) and transition to the first (respectively, second) type parameter $\kappa_{\mathbf{a}}$ (respectively, $\kappa_{\mathbf{b}}$). Intuitively, the type parameter $\kappa_{\mathbf{a}}$ would have the form $C^n[A]$ for $n > 0$ (respectively, $\kappa_{\mathbf{b}}$ would be $C^n[B]$). Then, the type $C[\kappa]$ would output an **R** and *pop* the symbol C from the stack by transitioning to κ. Once all the closing symbols have been outputted (note that you cannot terminate pre-emptively), we transition to type A or B depending on the intermediate symbol chosen. Type A outputs **a** and terminates, and similarly, type B outputs **b** and terminates. Thus, we simulate the L_3 language (not possible with CFSTs [51]) using two type parameters.

2.2.3 Concatenation of Dyck Words. We conclude this section by specifying some standard properties on balanced parentheses: *closure under concatenation* and *closure under wrapping*. If $w_1\$$ and $w_2\$$ are two balanced words, then so is $w_1 w_2\$$. Similarly, if $w\$$ is a balanced word, then so is $\mathbf{L}w\mathbf{R}\$$. These two properties can be proved by implementing *append* and *wrap* processes capturing the former and latter properties.

$$append : (w_1 : D_0), (w_2 : D_0) \vdash (w : D_0) \qquad wrap : (w : D_0) \vdash (w' : D_0)$$

The preceding declarations describe the type for the two processes. The *append* process uses two channels w_1 and w_2 of type D_0 and provides $w : D_0$, whereas *wrap* uses $w : D_0$ and provides $w' : D_0$. The actual implementations are described in Section 9.

3 DESCRIPTION OF TYPES

The underlying base system of session types is derived from a Curry-Howard interpretation [7, 8] of intuitionistic linear logic [27]. In the following, we describe the session types, their operational interpretation, and the continuation type.

Types A, B, C ::=	$\oplus\{\ell : A_\ell\}_{\ell \in L}$	send label $k \in L$	continue at type A_k
|	$\&\{\ell : A_\ell\}_{\ell \in L}$	receive label $k \in L$	continue at type A_k
|	$A \otimes B$	send channel of type A	continue at type B
|	$A \multimap B$	receive channel of type A	continue at type B
|	$\mathbf{1}$	send close message	no continuation
|	α	type variable	
|	$V[\theta]$	defined type name	

The basic type operators have the usual interpretation: the *internal choice* operator $\oplus\{\ell : A_\ell\}_{\ell \in L}$ selects a branch with label $k \in L$ with corresponding continuation type A_k; the *external choice* operator $\&\{\ell : A_\ell\}_{\ell \in L}$ offers a choice with labels $\ell \in L$ with corresponding continuation types A_ℓ; the *tensor* operator $A \otimes B$ represents the channel passing type that consists of sending a channel of type A and proceeding with type B; dually, the *lolli* operator $A \multimap B$ consists of receiving a channel of type A and continuing with type B; and the *terminated session* $\mathbf{1}$ is the operator that closes the session.

Types can also refer to parameter α available in scope. The *free variables* in type A refer to the set of type variables that occur freely in A.[1] Types without any free variables are called *closed types*.

We also support *type constructors* to define new *type names*. A type name V is defined according to a *type definition* $V[\overline{\alpha}] \triangleq A$ that is parameterized by a sequence of *distinct type variables* $\overline{\alpha}$ that the type A can refer to. The type $V[\theta]$ instantiates a type name with a substitution θ for the type parameters $\overline{\alpha}$.[2] We sometimes write $\mathcal{V}$ in place of $\overline{\alpha}$. Any type not of the form $V[\theta]$ is termed *structural*. For example, a type $V_1[\theta_1] \otimes V_2[\theta_2]$ is structural, and so are type variables α.

A substitution is a function from type variables to types.

$$\begin{array}{ll} \textit{Variables} & \mathcal{V} ::= \cdot \mid \mathcal{V}, \alpha \\ \textit{Substitutions} & \theta, \sigma, \phi ::= \cdot \mid \theta, A/\alpha \end{array}$$

We use the standard judgment $\mathcal{V}' \vdash \theta : \mathcal{V}$, which is defined in Section 6, to describe a substitution θ's domain (the type variables $\mathcal{V}$) and codomain (types whose free variables are contained in $\mathcal{V}'$). The types in the image of a substitution θ can involve type name instantiations of the form $U[\sigma]$, which means that a type expression of the form $V[\theta]$ can have *nested* type names. We represent the application of substitution θ to a type A by $\theta(A)$. The composition of substitutions is defined as the pointwise extension of the application of substitutions on types. Last, we say a substitution θ is $\mathcal{V}$*-closing* if $\cdot \vdash \theta : \mathcal{V}$.

All type definitions are stored in a finite global *signature* Σ defined as follows.

$$\textit{Signatures} \quad \Sigma ::= \cdot \mid \Sigma, V[\overline{\alpha}] \triangleq A$$

In a *valid signature*, all definitions $V[\overline{\alpha}] \triangleq A$ are contractive, meaning that A is *structural* and not a type variable. Most importantly, this means that A is not a type name instantiation. In particular, we do not allow the programmer to write definitions $V[\overline{\alpha}] \triangleq \alpha$ in signatures, but a programmer would not naturally write such definitions, instead writing $\theta(\alpha)$ anywhere that $V[\theta]$ would be written for such a V. The free variables occurring in A must be contained in $\overline{\alpha}$. This top-level scoping of all type variables is what we call the *prenex form of polymorphism*. We take an *equirecursive* view of type definitions, which means that unfolding a type definition does not require communication. More concretely, the type $V[\theta]$ is considered equivalent to its unfolding $\theta(A)$. We can easily adapt our definitions to an *isorecursive* view [21, 39] with explicit unfold messages. All type names V occurring in a valid signature must be defined, and all type variables defined in a valid definition must be distinct.

In addition, because we take an equirecursive view of type definitions and because type definitions have no free variables—only those bound by the type name's definition—we are justified in treating $\phi(V[\theta])$ and $V[\phi \circ \theta]$ as syntactically equal.

4 TYPE EQUALITY

Central to any practical type checking algorithm is type equality. In our system, it is necessary for the rule of identity (forwarding) and process spawn, as well as the channel-passing constructs for types $A \otimes B$ and $A \multimap B$. However, with nested polymorphic recursion, checking equality becomes challenging. We first develop the underlying theory of equality providing its definition, then establish its reduction to checking trace equivalence of deterministic first-order grammars.

[1] There are no type variable binders within types A. Instead, type *definitions* are parameterized by a sequence of type variables that function as binders, as described in the following paragraph.

[2] In the preceding section's examples, we used a more concrete, application-like syntax of, for instance, $V[A, B]$. When working with the theory, however, it is more convenient to cast this as instantiation by an explicit substitution so that, for example, $V[A, B]$ would instead be written as $V[A/\alpha, B/\beta]$ (or as $V[\theta]$, where $\theta = A/\alpha, B/\beta$).

4.1 Type Equality Definition

Intuitively, two types are equal if they permit exactly the *same* communication behavior. This reduces to checking if the next communication behavior that the types permit are equal, and the continuation types after the communication are equal as well. Informally, two communication behaviors are considered equal if it involves sending (respectively, receiving) the same labels, close message, or channels of equal types.

Formally, type equality is captured using a coinductive definition following the seminal work by Gay and Hole [25].

Definition 4.1. We define $\mathsf{unfold}_\Sigma(A)$ with respect to a signature Σ containing type definitions as follows.

$$\frac{V[\overline{\alpha}] \triangleq A \in \Sigma \quad \mathsf{unfold}_\Sigma(\theta(A)) = B}{\mathsf{unfold}_\Sigma(V[\theta]) = B}\ \mathsf{def} \qquad\qquad \frac{A \neq V[\theta]}{\mathsf{unfold}_\Sigma(A) = A}\ \mathsf{str}$$

We have a structural equirecursive type system where a type definition $V[\overline{\alpha}] \triangleq A$ enables $V[\theta]$ to be considered as $\theta(A)$. The function unfold is used to *unfold* type names. For this reason, unfolding a structural type simply returns the type itself. Since type definitions are *contractive* [25], unfolding always terminates. Even in the presence of type name definitions $V[\overline{\alpha}] \triangleq \alpha$ that will eventually be permitted in Section 5, unfolding always terminates. This is made precise by the following lemma. The key observation is that when $V[\overline{\alpha}] \triangleq \alpha$, we have $\mathsf{unfold}_\Sigma(V[\theta]) = B$ if and only if $\mathsf{unfold}_\Sigma(\theta(\alpha)) = B$ and $\theta(\alpha)$ is a proper syntactic subterm of the type $V[\theta]$ so that the type to be unfolded always becomes smaller. Thus, despite not being contractive in a strictly syntactic sense, definitions $V[\overline{\alpha}] \triangleq \alpha$ are unproblematic in the sense of a terminating $\mathsf{unfold}_\Sigma(-)$.

Moreover, due to its definition, unfolding any type always returns a structural type; if the type is closed, its unfolding is a structural type that is not a variable. (Incidentally, unfolding $V[\theta]$ into (the unfolding of) $\theta(A)$ in the first of the preceding rules is why we prefer the instantiation-based syntax over the application-based syntax when working with the theory.)

Lemma 4.2. *The* $\mathsf{unfold}_\Sigma(-)$ *operation always terminates in a structural type.*

Proof. By structural induction on the syntactic structure of type being unfolded.

If the type being unfolded is either a structural type or type variable, unfolding clearly terminates. In the other cases, the type being unfolded has the form $V[\theta]$; there are two possibilities:

Case: Consider the case in which $V[\overline{\alpha}] \triangleq A$ for some structural type A that is *not* a type variable. In this case, the type $\theta(A)$ is also a structural type (and not a type variable, as it happens), so $\mathsf{unfold}_\Sigma(\theta(A))$ terminates in a structural type, namely $\theta(A)$ itself. Because $\mathsf{unfold}_\Sigma(V[\theta]) = B$ if and only if $\mathsf{unfold}_\Sigma(\theta(A)) = B$, it follows that $\mathsf{unfold}_\Sigma(V[\theta])$ terminates in a structural type.

Case: Consider the case in which $V[\overline{\alpha}] \triangleq \alpha$. (Such definitions are not permitted in this section, but they will eventually be allowed in Section 5.) The type $\theta(\alpha)$ is a proper syntactic subterm of $V[\theta]$, so by the inductive hypothesis, $\mathsf{unfold}_\Sigma(\theta(\alpha))$ terminates in a structural type. (Here it is important that we work with the *syntactic* structure of terms. If we worked equirecursively up to unfolding in this induction, $\theta(\alpha)$ would be the same term as $V[\theta]$, not a smaller term.) Because $\mathsf{unfold}_\Sigma(V[\theta]) = B$ if and only if $\mathsf{unfold}_\Sigma(\theta(\alpha)) = B$, it follows that $\mathsf{unfold}_\Sigma(V[\theta])$ also terminates in a structural type. □

Definition 4.3. A symmetric binary relation $\mathcal{R}$ on *closed* types (no free variables) is said to *progress* to another symmetric binary relation $\mathcal{S}$ on closed types if $(A, B) \in \mathcal{R}$ implies the following:

- If $\mathsf{unfold}_\Sigma(A) = \oplus\{\ell : A_\ell\}_{\ell \in L}$, then $\mathsf{unfold}_\Sigma(B) = \oplus\{\ell : B_\ell\}_{\ell \in L}$ and also $(A_\ell, B_\ell) \in \mathcal{S}$ for all $\ell \in L$.

Table 1. Actions and Continuation Types

Closed Type's Unfolding	Action(s)	Continuation Type(s)
$\oplus\{\ell : A_\ell\}_{\ell \in L}$	$\oplus k$, for all $k \in L$	A_k
$\&\{\ell : A_\ell\}_{\ell \in L}$	$\& k$, for all $k \in L$	A_k
$A \otimes B$	$\otimes_1$ and $\otimes_2$	A and B, respectively
$A \multimap B$	$\multimap_1$ and $\multimap_2$	A and B, respectively
$\mathbf{1}$	$\mathbf{1}$	ϵ

- If $\mathsf{unfold}_\Sigma(A) = \&\{\ell : A_\ell\}_{\ell \in L}$, then $\mathsf{unfold}_\Sigma(B) = \&\{\ell : B_\ell\}_{\ell \in L}$ and also $(A_\ell, B_\ell) \in \mathcal{S}$ for all $\ell \in L$.
- If $\mathsf{unfold}_\Sigma(A) = A_1 \otimes A_2$, then $\mathsf{unfold}_\Sigma(B) = B_1 \otimes B_2$ and $(A_1, B_1) \in \mathcal{S}$ and $(A_2, B_2) \in \mathcal{S}$.
- If $\mathsf{unfold}_\Sigma(A) = A_1 \multimap A_2$, then $\mathsf{unfold}_\Sigma(B) = B_1 \multimap B_2$ and $(B_1, A_1) \in \mathcal{S}$ and $(A_2, B_2) \in \mathcal{S}$.
- If $\mathsf{unfold}_\Sigma(A) = \mathbf{1}$, then $\mathsf{unfold}_\Sigma(B) = \mathbf{1}$.

A binary relation $\mathcal{R}$ on closed types that progresses to itself is called a *type bisimulation*.

Definition 4.4. Two closed types A and B are equal (written $\vDash A \equiv_\Sigma B$) if and only if there exists a type bisimulation $\mathcal{R}$ such that $(A, B) \in \mathcal{R}$. (The subscript Σ is usually omitted because the signature is virtually always apparent from the context.)

These definitions of progression and type bisimulation implicitly describe a labeled transition system. The labels, or *actions*, in that system correspond to the structural type operator that appears in a closed type's unfolding. For each action that a type can take, there is a unique *continuation type* to which it evolves. For example, the actions that the type $\oplus\{\ell : A_\ell\}_{\ell \in L}$ could take might be written as $\oplus k$, for all $k \in L$, and would have the continuation types A_k, respectively. In other words, if the labeled transition system were to made explicit, there would be transitions $\oplus\{\ell : A_\ell\}_{\ell \in L} \xrightarrow{\oplus k} A_k$ for all $k \in L$.

Table 1 displays the actions and corresponding continuation types for closed structural types that form the basis of the labeled transition system implicit in the preceding definitions.

We choose not to make the labeled transition system explicit because we feel its details mostly obscure the straightforward intuition behind the definitions of progression and type bisimulations. That $\mathcal{R}$ progresses to $\mathcal{S}$ means that $(A, B) \in \mathcal{R}$ implies that (the unfoldings of) types A and B have matching actions and $\mathcal{S}$-related continuation types. We will rely on this kind of intuitive reading of progression in the proofs found in Section 5.

This definition only applies to types with no free type variables. Since we allow parameters in type definitions, we need to define equality in the presence of free type variables. To this end, we define the notation $\mathcal{V} \vDash A \equiv B$, where $\mathcal{V}$ is a collection of type variables and A and B are types whose free variables are contained in $\mathcal{V}$. Equality of types with free variables is defined in terms of the equality of all closed instances of those types. We have the following definition.

Definition 4.5. $\mathcal{V} \vDash A \equiv B$ holds if and only if $\vDash \theta(A) \equiv \theta(B)$ for all $\mathcal{V}$-closing substitutions θ.

4.2 Decidability of Type Equality

Solomon [48] proved that types defined using parametric type definitions with an *inductive interpretation* can be translated to DPDAs, thus reducing type equality to language equality on DPDAs. However, our type definitions have a *coinductive interpretation*. As an example, consider the types $A = \oplus\{\mathbf{a} : A\}$ and $B = \oplus\{\mathbf{b} : B\}$. With an *inductive* interpretation, types A and B are empty (because they do not have terminating symbols) and thus are equal. However, with a *coinductive* interpretation, type A will send an infinite number of $\mathbf{a}$'s, and B will send an infinite number of $\mathbf{b}$'s, and are thus not equal. Our reduction needs to account for this coinductive behavior.

We show that type equality of nested session types is decidable via a reduction to the trace equivalence problem of deterministic first-order grammars [33]. A *first-order grammar* is a structure $(\mathcal{N}, \mathcal{A}, \mathcal{S})$, where $\mathcal{N}$ is a set of non-terminals, $\mathcal{A}$ is a finite set of *actions*, and $\mathcal{S}$ is a finite set of *production rules*. The arity of non-terminal $N \in \mathcal{N}$ is written as $\mathrm{arity}(N) \in \mathbb{N}$. Production rules rely on a countable set of *variables* $\mathcal{X}$ and on the set $\mathcal{T}_{\mathcal{N}}$ of *regular terms* over $\mathcal{N} \cup \mathcal{X}$. A term is *regular* if the set of subterms is finite (see [33]). Similarly to our types, regular terms can be infinite but have finite representations; intuitively, regular terms can be interpreted as the unfolding of finite graphs.

Each production rule has the form $N\overline{\alpha} \xrightarrow{a} E$, where $N \in \mathcal{N}$ is a non-terminal, $a \in \mathcal{A}$ is an action, and $\overline{\alpha} \in \mathcal{X}^*$ are variables that the term $E \in \mathcal{T}_{\mathcal{N}}$ can refer to. A grammar is *deterministic* if for each pair of $N \in \mathcal{N}$ and $a \in \mathcal{A}$ there is at most one rule of the form $N\overline{\alpha} \xrightarrow{a} E$ in $\mathcal{S}$. The substitution of terms $\overline{B}$ for variables $\overline{\alpha}$ in a rule $N\overline{\alpha} \xrightarrow{a} E$, denoted by $N\overline{B} \xrightarrow{a} E[\overline{B}/\overline{\alpha}]$, is the rule $(N\overline{\alpha} \xrightarrow{a} E)[\overline{B}/\overline{\alpha}]$. Given a set of rules $\mathcal{S}$, the trace of a term T is defined as $\mathrm{trace}_{\mathcal{S}}(T) = \{\overline{a} \in \mathcal{A}^* \mid (T \xrightarrow{\overline{a}} T') \in \mathcal{S},$ for some $T' \in \mathcal{T}_{\mathcal{N}}\}$. Two terms are *trace equivalent*, written as $T \sim_{\mathcal{S}} T'$, if $\mathrm{trace}_{\mathcal{S}}(T) = \mathrm{trace}_{\mathcal{S}}(T')$.

The crux of the reduction lies in the observation that session types can be translated to terms and type definitions can be translated to production rules of a first-order grammar. We start the translation of nested session types to grammars by first making an initial pass over the signature and introducing fresh *internal names* such that the new type definitions alternate between structural and non-structural types. These internal names are parameterized over their free type variables, and their definitions are added to the signature. This intermediate representation simplifies the next step where we translate this extended signature to grammar production rules.

The *internal renaming* is defined using the judgment $\Sigma \rightsquigarrow \Sigma'$ as described in Figure 1. An empty signature renames to itself as described in the emp rule. The step rule describes taking a definition $(V[\overline{\alpha}] \triangleq A)$ from Σ, and renaming A to B and adding definition $(V[\overline{\alpha}] \triangleq B)$ to the renamed signature Σ'. Choosing $\mathcal{V} = \overline{\alpha}$, this process of renaming types is defined using two mutually recursive judgments $\mathcal{V} \vdash A \Rightarrow (B \; ; \; \Sigma)$ and $\mathcal{V} \vdash A \rightarrow (B \; ; \; \Sigma)$, renaming type A to type B and introducing fresh internal type definitions collected in signature Σ. We need two distinct judgments to recognize the alternating behavior of renaming: fresh names are only introduced alternately to generate *contractive* type definitions. The former judgment is responsible for generating a fresh name on encountering a structural type with a continuation (first premise in rule $S_{\Rightarrow}$). It renames A to B (using the latter judgment) and generates definition $X[\overline{\alpha}] \triangleq B$ and adds it to signature Σ. If it encounters a defined type name (rule $N_{\Rightarrow}$), it renames its type parameters θ. Type variables are not renamed ($\mathrm{var}_{\Rightarrow}$). The latter judgment does not generate a fresh name but simply case analyzes on the structure of A and calls the former judgment on the continuation types (rules $\oplus_{\rightarrow}$, $\&_{\rightarrow}$, $\otimes_{\rightarrow}$, $\multimap_{\rightarrow}$). The rules $\mathbf{1}_{\rightarrow}$ and $\mathrm{var}_{\rightarrow}$ simply terminate by returning the input with an empty signature since they do not have continuations. The alternating nature of renaming is reflected in our rules with each judgment calling the other one. The $\theta_{\Rightarrow}$ rule extends the $\mathcal{V} \vdash A \Rightarrow (B \; ; \; \Sigma)$ judgment to substitutions in a pointwise fashion by calling the latter judgment on each type in the substitution. The following example elucidates this internal renaming procedure.

Example 4.6. As a running example, consider the following queue type from Section 2.

$$\mathrm{Queue}[\alpha] \triangleq \&\{\mathbf{enq} : \alpha \multimap \mathrm{Queue}[\alpha], \mathbf{deq} : \oplus\{\mathbf{none} : \mathbf{1}, \mathbf{some} : \alpha \otimes \mathrm{Queue}[\alpha]\}\}$$

(In the examples, we write $V[\theta]$ as $V[A_1, \ldots, A_n]$, for $\theta = (A_1/\alpha_1, \ldots, A_n/\alpha_n)$.)

$$\frac{\mathcal{V} \vdash A_\ell \Rightarrow (B_\ell \;;\; \Sigma_\ell) \quad (\forall \ell \in L)}{\mathcal{V} \vdash \oplus\{\ell : A_\ell\}_{\ell \in L} \rightarrow (\oplus\{\ell : B_\ell\}_{\ell \in L} \;;\; \cup_{\ell \in L}\Sigma_\ell)} \oplus_{\rightarrow} \qquad \frac{\mathcal{V} \vdash A_\ell \Rightarrow (B_\ell \;;\; \Sigma_\ell) \quad (\forall \ell \in L)}{\mathcal{V} \vdash \&\{\ell : A_\ell\}_{\ell \in L} \rightarrow (\&\{\ell : B_\ell\}_{\ell \in L} \;;\; \cup_{\ell \in L}\Sigma_\ell)} \&_{\rightarrow}$$

$$\frac{\mathcal{V} \vdash A_1 \Rightarrow (B_1 \;;\; \Sigma_1) \qquad \mathcal{V} \vdash A_2 \Rightarrow (B_2 \;;\; \Sigma_2)}{\mathcal{V} \vdash A_1 \otimes A_2 \rightarrow (B_1 \otimes B_2 \;;\; \Sigma_1, \Sigma_2)} \otimes_{\rightarrow} \qquad \frac{\mathcal{V} \vdash A_1 \Rightarrow (B_1 \;;\; \Sigma_1) \qquad \mathcal{V} \vdash A_2 \Rightarrow (B_2 \;;\; \Sigma_2)}{\mathcal{V} \vdash A_1 \multimap A_2 \rightarrow (B_1 \multimap B_2 \;;\; \Sigma_1, \Sigma_2)} \multimap_{\rightarrow}$$

$$\frac{}{\mathcal{V} \vdash \mathbf{1} \rightarrow (\mathbf{1} \;;\; \cdot)} \mathbf{1}_{\rightarrow} \qquad \frac{}{\mathcal{V} \vdash \alpha \rightarrow (\alpha \;;\; \cdot)} \mathsf{var}_{\rightarrow} \qquad \frac{}{\mathcal{V} \vdash \alpha \Rightarrow (\alpha \;;\; \cdot)} \mathsf{var}_{\Rightarrow}$$

$$\frac{A \neq V[\theta], \alpha \qquad \mathcal{V} \vdash A \rightarrow (B \;;\; \Sigma) \qquad (X \text{ fresh}, \mathcal{V} = \overline{\alpha})}{\mathcal{V} \vdash A \Rightarrow (X[\overline{\alpha}] \;;\; \Sigma, X[\overline{\alpha}] \triangleq B)} S_{\Rightarrow} \qquad \frac{\mathcal{V} \vdash \theta \Rightarrow (\theta' \;;\; \Sigma')}{\mathcal{V} \vdash V[\theta] \Rightarrow (V[\theta'] \;;\; \Sigma')} N_{\Rightarrow}$$

$$\frac{}{\mathcal{V} \vdash (\cdot) \Rightarrow (\cdot \;;\; \cdot)} \mathsf{emp}_{\Rightarrow} \qquad \frac{\mathcal{V} \vdash \theta \Rightarrow (\theta' \;;\; \Sigma') \qquad \mathcal{V} \vdash A \Rightarrow (B \;;\; \Sigma_0)}{\mathcal{V} \vdash \theta, A/\alpha \Rightarrow (\theta', B/\alpha \;;\; \Sigma', \Sigma_0)} \theta_{\Rightarrow}$$

$$\frac{}{(\cdot) \rightsquigarrow (\cdot)} \mathsf{emp} \qquad \frac{\Sigma \rightsquigarrow \Sigma' \quad \overline{\alpha} \vdash A \rightarrow (B \;;\; \Sigma_A)}{\Sigma, V[\overline{\alpha}] \triangleq A \rightsquigarrow \Sigma', \Sigma_A, V[\overline{\alpha}] \triangleq B} \mathsf{step}$$

Fig. 1. Algorithmic rules for internal renaming.

After performing internal renaming for this type, we obtain the following signature.

$$\mathrm{Queue}[\alpha] \triangleq \&\{\mathbf{enq} : X_0[\alpha], \mathbf{deq} : X_1[\alpha]\} \qquad X_0[\alpha] \triangleq \alpha \multimap \mathrm{Queue}[\alpha]$$

$$X_1[\alpha] \triangleq \oplus\{\mathbf{none} : X_2[\alpha], \mathbf{some} : X_3[\alpha]\} \qquad X_2[\alpha] \triangleq \mathbf{1} \qquad X_3[\alpha] \triangleq \alpha \otimes \mathrm{Queue}[\alpha]$$

We introduce the fresh internal names X_0, X_1, X_2, and X_3 (parameterized with free variable α) to represent the continuation type in each case. Note the alternation between structural and non-structural types.

Next, we translate this extended signature to the grammar $\mathcal{G} = (\mathcal{N}, \mathcal{A}, \mathcal{S})$ aimed at reproducing the behavior prescribed by the types as grammar actions.

$$\begin{array}{rcl}
\mathcal{N} & = & \{\mathrm{Queue}, X_0, X_1, X_2, X_3, \bot\} \\
\mathcal{A} & = & \{\&\mathbf{enq}, \&\mathbf{deq}, \multimap_1, \multimap_2\ \oplus\mathbf{none}, \oplus\mathbf{some}, \otimes_1, \otimes_2, \mathbf{1}\} \\
\mathcal{S} & = & \{\mathrm{Queue}\,\alpha \xrightarrow{\&\mathbf{enq}} X_0\alpha,\ \ \mathrm{Queue}\,\alpha \xrightarrow{\&\mathbf{deq}} X_1\alpha,\ \ X_0\alpha \xrightarrow{\multimap_1} \alpha,\ \ X_0\alpha \xrightarrow{\multimap_2} \mathrm{Queue}\,\alpha, \\
& & \ X_1\alpha \xrightarrow{\oplus\mathbf{none}} X_2\alpha,\ \ X_1\alpha \xrightarrow{\oplus\mathbf{some}} X_3\alpha,\ \ X_2\alpha \xrightarrow{\mathbf{1}} \bot,\ \ X_3\alpha \xrightarrow{\otimes_1} \alpha,\ \ X_3\alpha \xrightarrow{\otimes_2} \mathrm{Queue}\,\alpha\}
\end{array}$$

Essentially, each defined type name is translated into a fresh non-terminal. Each type definition then corresponds to a sequence of production rules: one for each possible continuation type with the appropriate label that leads to that continuation. For instance, the type $\mathrm{Queue}[\alpha]$ has two possible continuations: transition to $X_0[\alpha]$ with action $\&\mathbf{enq}$ or to $X_1[\alpha]$ with action $\&\mathbf{deq}$. The rules for all other type names are analogous. When the continuation is $\mathbf{1}$, we create a fresh non-terminal that transitions through action $\mathbf{1}$ to the nullary non-terminal $\bot$, disabling any further action. When the continuation is α, we transition to α. Since each type name is defined once, the produced grammar is deterministic.

Formally, the translation from an (extended) signature to a grammar is handled by two simultaneous tasks: translating type definitions into production rules (function τ below), and converting type names and variables into grammar terms (function $(\!|\cdot|\!)$). The function $(\!|\cdot|\!)$ is defined by the following.

$$\begin{array}{rcll}
(\!|\alpha|\!) & = & \alpha & \text{type variables translate to themselves} \\
(\!|V[\theta]|\!) & = & V(\!|B_1|\!) \cdots (\!|B_n|\!),\ \text{for } \theta = (B_1/\alpha_1, \ldots, B_n/\alpha_n) & \text{type names translate to grammar terms}
\end{array}$$

Due to this mapping, throughout this section we will use type names indistinctly as type names or as non-terminal first-order symbols.

The function τ converts a type definition $V[\overline{\alpha}] \triangleq A$ into a set of production rules and is defined according to the structure of A as follows.

$$\begin{array}{rcl}
\tau(V[\overline{\alpha}] \triangleq \oplus\{\ell : A_\ell\}_{\ell \in L}) & = & \{(\!|V[\overline{\alpha}]|\!) \xrightarrow{\oplus\ell} (\!|A_\ell|\!) \mid \ell \in L\} \\
\tau(V[\overline{\alpha}] \triangleq \&\{\ell : A_\ell\}_{\ell \in L}) & = & \{(\!|V[\overline{\alpha}]|\!) \xrightarrow{\&\ell} (\!|A_\ell|\!) \mid \ell \in L\} \\
\tau(V[\overline{\alpha}] \triangleq A_1 \otimes A_2) & = & \{(\!|V[\overline{\alpha}]|\!) \xrightarrow{\otimes_i} (\!|A_i|\!) \mid i = 1, 2\} \\
\tau(V[\overline{\alpha}] \triangleq A_1 \multimap A_2) & = & \{(\!|V[\overline{\alpha}]|\!) \xrightarrow{\multimap_i} (\!|A_i|\!) \mid i = 1, 2\} \\
\tau(V[\overline{\alpha}] \triangleq \mathbf{1}) & = & \{(\!|V[\overline{\alpha}]|\!) \xrightarrow{\mathbf{1}} \bot\}
\end{array}$$

Function τ identifies the actions and continuation types corresponding to A and translates them into grammar rules. Internal and external choices lead to actions $\oplus\ell$ and $\&\ell$, for each $\ell \in L$, with A_ℓ as the continuation type. The type $A_1 \otimes A_2$ enables two possible actions, $\otimes_1$ and $\otimes_2$, with continuation A_1 and A_2, respectively. Similarly $A_1 \multimap A_2$ produces the actions $\multimap_1$ and $\multimap_2$ with A_1 and A_2 as respective continuations. The terminated session $\mathbf{1}$ enables the action with the same name $\mathbf{1}$ with continuation $\bot$, avoiding any further actions. Contractiveness ensures that there are no definitions of the form $V[\overline{\alpha}] \triangleq V'[\theta]$. Our internal renaming ensures that we do not encounter cases of the form $V[\overline{\alpha}] \triangleq \alpha$ because we do not generate internal names for variables. For this reason, the $(\!|\cdot|\!)$ function is only defined on the complement types α and $V[\theta]$.

The τ function is extended to translate a signature pointwise. Formally, that is, $\tau(\Sigma) = \bigcup_{(V[\overline{\alpha}] \triangleq A) \in \Sigma} \tau(V[\overline{\alpha}] \triangleq A)$. Then connecting all of these pieces, we finally define the fog function that translates a signature to a grammar as follows.

$$\mathrm{fog}(\Sigma) = (\mathcal{N}, \mathcal{A}, \mathcal{S}), \text{ where}$$
$$\mathcal{S} = \tau(\Sigma) \qquad \mathcal{N} = \{N \mid (N\overline{\alpha} \xrightarrow{a} E) \in \tau(\Sigma)\} \qquad \mathcal{A} = \{a \mid (N\overline{\alpha} \xrightarrow{a} E) \in \tau(\Sigma)\}$$

The grammar is constructed by first computing $\tau(\Sigma)$ to obtain all production rules. Then the sets $\mathcal{N}$ and $\mathcal{A}$ are constructed by collecting the set of non-terminals and actions from these production rules. The finite representation of session types and uniqueness of definitions ensure that the grammar $\mathrm{fog}(\Sigma)$ is, in fact, a deterministic first-order grammar.

Checking the equality of types A and B given a signature Σ finally reduces to (i) the internal renaming of Σ to produce Σ', and (ii) check the trace-equivalence of terms $(\!|A|\!)$ and $(\!|B|\!)$ given grammar $\mathrm{fog}(\Sigma')$. If A and B are themselves structural, we generate internal names for them during the internal renaming process. Since we assume an *equirecursive* and *non-generative* view of types, it is easy to show that internal renaming does not alter the communication behavior of types and preserves type equality, as formalized in the following lemma.

Lemma 4.7. *$\vDash A \equiv_\Sigma B$ if and only if $\vDash A \equiv_{\Sigma'} B$.*

Proof. The result follows by noting that signatures Σ and Σ' are equivalent. In a brief proof sketch, to prove that Σ and Σ' are equivalent, consider definitions $V[\overline{\alpha}] \triangleq C \in \Sigma$ and $V[\overline{\alpha}] \triangleq D \in \Sigma'$ and show that $\vDash C \equiv D$, where definitions on the left are expanded using signature Σ while definitions on the right are expanded using Σ'. □

Theorem 4.8. *$\vDash A \equiv_\Sigma B$ if and only if $(\!|A|\!) \sim_{\mathcal{S}} (\!|B|\!)$, where $(\mathcal{N}, \mathcal{A}, \mathcal{S}) = \mathrm{fog}(\Sigma')$ and Σ' is the extended signature for Σ.*

Proof. For the direct implication, assume that $(\!|A|\!) \nsim_{\mathcal{S}} (\!|B|\!)$. Pick a sequence of actions in the difference of the traces and let w_0 be its greatest prefix occurring in both traces, with $(\!|A|\!) \xrightarrow{w_0} (\!|A'|\!)$

and $(\!|B|\!) \xrightarrow{w_0} (\!|B'|\!)$. Either w_0 is a maximal trace for one of the terms or we have $(\!|A'|\!) \xrightarrow{a_1} (\!|A''|\!)$ and $(\!|B'|\!) \xrightarrow{a_2} (\!|B''|\!)$, where $a_1 \neq a_2$. In both cases, we have $\nvDash A' \equiv B'$. To show that, let us proceed by case analysis on A' assuming that $\vDash A' \equiv B'$. We have the following cases:

Case $\mathsf{unfold}_\Sigma(A') = \oplus\{\ell : A_\ell\}_{\ell \in L}$. In this case, we would have $\mathsf{unfold}_\Sigma(B') = \oplus\{\ell : B_\ell\}_{\ell \in L}$. Hence, we would have $a_1 = \oplus\ell$ for some $\ell \in L$ and $w = w_0 \cdot a_1$ would occur in both traces and would be greater than w_0, which is a contradiction.

Case $\mathsf{unfold}_\Sigma(A') = \&\{\ell : A_\ell\}_{\ell \in L}$. Similar to the previous case.

Case $\mathsf{unfold}_\Sigma(A') = A_1 \otimes A_2$. In this case we would have $\mathsf{unfold}_\Sigma(B') = B_1 \otimes B_2$. Hence, $a_1 \in \{\otimes_1, \otimes_2\}$ and we would have $w = w_0 \cdot a_1$ occurring in both traces, which contradicts the assumption that w_0 is the greatest prefix occurring in both traces.

Case $\mathsf{unfold}_\Sigma(A') = A_1 \multimap A_2$. Similar to the previous case.

Case $\mathsf{unfold}_\Sigma(A') = \mathbf{1}$. In this case, we would have $\mathsf{unfold}_\Sigma(B') = \mathbf{1}$. Hence, we would have $a_1 = \mathbf{1}$ and $w_0 \cdot a_1$ would occur in both traces and, again, would be greater than w_0.

Since all cases led to contradictions, we have $\nvDash A' \equiv B'$. The conclusion that $\nvDash A \equiv B$ follows immediately from the this property: if $(\!|A_0|\!) \xrightarrow{w} (\!|A_1|\!)$ and $(\!|B_0|\!) \xrightarrow{w} (\!|B_1|\!)$ and $A_1 \not\equiv B_1$, then $A_0 \not\equiv B_0$. We prove this property by induction on the length of w. If $|w| = 0$, then A_1 coincides with A_0 and B_1 coincides with B_0, so $\nvDash A_0 \equiv B_0$. Now, let $n > 0$ and assume the property holds for any trace of length n. Consider $w = w' \cdot a$ with $|w'| = n$ and let A_2, B_2 be subject to $(\!|A_0|\!) \xrightarrow{w'} (\!|A_2|\!) \xrightarrow{a} (\!|A_1|\!)$ and $(\!|B_0|\!) \xrightarrow{w'} (\!|B_2|\!) \xrightarrow{a} (\!|B_1|\!)$. With a case analysis on A_2, similar to the preceding analysis, since $\nvDash A_1 \equiv B_1$, we conclude that $\nvDash A_2 \equiv B_2$. By the induction hypothesis, we have $\nvDash A_0 \equiv B_o$.

For the reciprocal implication, assume that $(\!|A|\!) \sim_\mathcal{S} (\!|B|\!)$. Consider the relation

$$\mathcal{R} = \{(A_0, B_0) \mid \mathsf{trace}_\mathcal{S}((\!|A_0|\!)) = \mathsf{trace}_\mathcal{S}((\!|B_0|\!))\} \cup \{(\bot, \bot)\}.$$

Obviously, $(A, B) \in \mathcal{R}$. To prove that $\mathcal{R}$ is a type bisimulation, let $(A_0, B_0) \in \mathcal{R}$ and proceed by case analysis on A_0 and B_0. We sketch a couple of cases for A_0. The other cases are analogous. Two representative cases are:

Case $\mathsf{unfold}_\Sigma(A_0) = \oplus\{\ell : A_\ell\}_{\ell \in L}$. In this case, we have $(\!|A_0|\!) \xrightarrow{\oplus\ell} (\!|A_\ell|\!)$. Since, by hypothesis, the traces coincide, $\mathsf{trace}_\mathcal{S}((\!|A_0|\!)) = \mathsf{trace}_\mathcal{S}((\!|B_0|\!))$, we have $(\!|B_0|\!) \xrightarrow{\oplus\ell} (\!|B_\ell|\!)$ and thus $\mathsf{unfold}_\Sigma(B_0) = \oplus\{\ell : B_\ell\}_{\ell \in L}$. Moreover, using Observation 3 of Jančar [33], we have $\mathsf{trace}_\mathcal{S}((\!|A_\ell|\!)) = \mathsf{trace}_\mathcal{S}((\!|B_\ell|\!))$. Hence, $(A_\ell, B_\ell) \in \mathcal{R}$.

Case $\mathsf{unfold}_\Sigma(A_0) = \mathbf{1}$. In this case, we have $(\!|A_0|\!) \xrightarrow{\mathbf{1}} \bot$ and $\mathsf{trace}_\mathcal{S}((\!|A_0|\!)) = \{\mathbf{1}\}$. Since $\mathsf{trace}_\mathcal{S}((\!|A_0|\!)) = \mathsf{trace}_\mathcal{S}((\!|B_0|\!))$, we have $\mathsf{unfold}_\Sigma(B_0) = \mathbf{1}$ and $(\!|B_0|\!) \xrightarrow{\mathbf{1}} \bot$. We conclude recalling that $(\bot, \bot) \in \mathcal{R}$. □

However, type equality is not only restricted to closed types (see Definition 4.5). To decide equality for open types (i.e., $\mathcal{V} \vDash A \equiv B$ given signature Σ), we introduce a fresh label ℓ_α and a nullary type name V_α for each $\alpha \in \mathcal{V}$. We extend the signature with type definitions: $\Sigma^* = \Sigma \cup_{\alpha \in \mathcal{V}} \{V_\alpha \triangleq \oplus\{\ell_\alpha : V_\alpha\}\}$. We then replace all occurrences of α in A and B with V_α and check their equality over signature Σ^*. We prove that this substitution preserves equality.

Theorem 4.9. *$\mathcal{V} \vDash A \equiv B$ if and only if $\vDash \sigma^*(A) \equiv_{\Sigma^*} \sigma^*(B)$, where $\sigma^*(\alpha) = V_\alpha$ for all $\alpha \in \mathcal{V}$.*

Proof. The direct implication is immediate because σ^* is a $\mathcal{V}$-closing substitution. For the reciprocal implication, assume that $\mathcal{V} \nvDash A \equiv B$. In this case, there exists σ' for which $\nvDash \sigma'(A) \equiv \sigma'(B)$. Using Theorem 4.8, we know that the traces for $(\!|\sigma'(A)|\!)$ and $(\!|\sigma'(B)|\!)$ are distinct. A sequence of actions w picked from the difference of the traces can either (i) be observed before reaching the substitution, in which case w is itself a witness that $(\!|\sigma^*(A)|\!) \nsim_\mathcal{S} (\!|\sigma^*(B)|\!)$, or (ii) result from the substitution. In the latter case, the greatest prefix for w belonging to both $\mathsf{trace}((\!|\sigma'(A)|\!))$ and trace($(\!|\sigma'(B)|\!)$), denoted by w_1, leads to a subterm C of $(\!|\sigma'(\beta)|\!)$ and to a subterm D of $(\!|\sigma'(\gamma)|\!)$, where $\beta \neq \gamma$: $(\!|\sigma'(A)|\!) \xrightarrow{w_1} C$ and $(\!|\sigma'(B)|\!) \xrightarrow{w_1} D$. In that case, there is a *subtrace* w_0 of w_1 such that $(\!|A|\!) \xrightarrow{w_0} \beta$ and $(\!|B|\!) \xrightarrow{w_0} \gamma$. Hence, we know that $(\!|\sigma^*(A)|\!) \xrightarrow{w_0} (\!|V_\beta|\!)$ and $(\!|\sigma^*(B)|\!) \xrightarrow{w_0} (\!|V_\gamma|\!)$ and $(\!|V_\beta|\!) \nsim_{\mathcal{S}} (\!|V_\gamma|\!)$, because ℓ_β and ℓ_γ are distinct labels. We conclude that $(\!|\sigma^*(A)|\!) \nsim_{\mathcal{S}} (\!|\sigma^*(B)|\!)$, and thus, using Theorem 4.8, we have $\nvDash \sigma^*(A) \equiv_{\Sigma^*} \sigma^*(B)$. □

THEOREM 4.10. *Checking $\mathcal{V} \vDash A \equiv B$ is decidable.*

PROOF. Theorem 4.9 reduces equality of open types to equality of closed types. Theorem 4.8 reduces equality of closed nested session types to trace equivalence of first-order grammars. Jančar [33] proved that trace equivalence for first-order grammars is decidable, hence establishing the decidability of equality for nested session types. □

5 PRACTICAL ALGORITHM FOR TYPE EQUALITY

Although type equality can be reduced to trace equivalence for first-order grammars (Theorem 4.8 and Theorem 4.9), the latter problem has a very high theoretical complexity with no known practical algorithm [33]. In response, we have designed a coinductive algorithm for approximating type equality. Taking inspiration from Gay and Hole [25], we attempt to construct a bisimulation. Our proposed algorithm is sound but incomplete and can terminate in three states: (i) types are proved equal by (implicitly) constructing a bisimulation, (ii) counterexample detected by identifying a position where types differ, or (iii) terminated without a conclusive answer due to incompleteness. We interpret both (ii) and (iii) as a failure of type checking (but there is a recourse; see Section 5.1). The algorithm is deterministic (no backtracking), and the implementation is quite efficient in practice. For all our examples, type checking is instantaneous (see Section 8).

The fundamental operation in the equality algorithm is *loop detection*, where we determine if we have already added an equation $A \equiv B$ to the bisimulation we are constructing. Due to the presence of *open types* with free type variables, determining if we have already considered an equation becomes a difficult operation. To that purpose, we make an initial pass over the given types and introduce fresh internal names as described in Figure 1 but, for simplicity, also renaming variables α by eliminating the $\mathrm{var}_\Rightarrow$ rule and changing the first premise of rule $S_\Rightarrow$ to just $A \neq V[\theta]$. This results in a kind of *strict normal form*, in contrast to the slightly more relaxed normal form of the previous section that did not require renaming of variables.[3] In the resulting signature, defined type names and structural types alternate, as in the following grammars. Also notice that substitutions now do not involve structural types that are not variables.

$$
\begin{aligned}
A, B, C &::= \oplus\{\ell : V[\theta_\ell]\}_{\ell \in L} \mid \&\{\ell : V[\theta_\ell]\}_{\ell \in L} \mid V_1[\theta_1] \otimes V_2[\theta_2] \mid V_1[\theta_1] \multimap V_2[\theta_2] \mid \mathbf{1} \mid \alpha \\
\theta, \sigma, \phi &::= (\cdot) \mid \theta, (V[\theta']/\alpha) \mid \theta, (\beta/\alpha) \\
\Sigma &::= (\cdot) \mid \Sigma, V[\overline{\alpha}] \triangleq A
\end{aligned}
$$

In the preceding sections, we used metavariables A, B, and C to characterize all types. From this point to the end of the current section, we will use A, B, and C as given in the preceding grammar. We could introduce more metavariables, but that would make the notation heavier.

With this additional structure on types, we can perform loop detection entirely on defined type names (whether internal or external). Based on the invariants established by internal names, the algorithm never needs to compare type name instantiations with non-variable structural types.

[3]Definitions $V[\overline{\alpha}] \triangleq \alpha$, which are not permitted in programmer-written signatures, do arise under this stricter internal renaming. However, they are unproblematic because they allow a terminating $\mathrm{unfold}_\Sigma(-)$ (see Lemma 4.2), even though they are not contractive in a strictly syntactic sense.

$$\frac{\Psi\ ;\ \mathcal{V} \vdash V_\ell[\theta_\ell] \equiv U_\ell[\sigma_\ell] \quad (\forall \ell \in L)}{\Psi\ ;\ \mathcal{V} \vdash \oplus\{\ell : V_\ell[\theta_\ell]\}_{\ell \in L} \equiv \oplus\{\ell : U_\ell[\sigma_\ell]\}_{\ell \in L}}\ \oplus \qquad \frac{\Psi\ ;\ \mathcal{V} \vdash V_\ell[\theta_\ell] \equiv U_\ell[\sigma_\ell] \quad (\forall \ell \in L)}{\Psi\ ;\ \mathcal{V} \vdash \&\{\ell : V_\ell[\theta_\ell]\}_{\ell \in L} \equiv \&\{\ell : U_\ell[\sigma_\ell]\}_{\ell \in L}}\ \&$$

$$\frac{\Psi\ ;\ \mathcal{V} \vdash V_1[\theta_1] \equiv U_1[\sigma_1] \quad \Psi\ ;\ \mathcal{V} \vdash V_2[\theta_2] \equiv U_2[\sigma_2]}{\Psi\ ;\ \mathcal{V} \vdash V_1[\theta_1] \otimes V_2[\theta_2] \equiv U_1[\sigma_1] \otimes U_2[\sigma_2]}\ \otimes \qquad \frac{}{\Psi\ ;\ \mathcal{V} \vdash \mathbf{1} \equiv \mathbf{1}}\ \mathbf{1}$$

$$\frac{\Psi\ ;\ \mathcal{V} \vdash V_1[\theta_1] \equiv U_1[\sigma_1] \quad \Psi\ ;\ \mathcal{V} \vdash V_2[\theta_2] \equiv U_2[\sigma_2]}{\Psi\ ;\ \mathcal{V} \vdash V_1[\theta_1] \multimap V_2[\theta_2] \equiv U_1[\sigma_1] \multimap U_2[\sigma_2]}\ \multimap$$

$$\frac{\alpha \in \mathcal{V}}{\Psi\ ;\ \mathcal{V} \vdash \alpha \equiv \alpha}\ \text{v-v} \qquad \frac{\alpha \in \mathcal{V} \quad \text{unfold}_\Sigma(U[\sigma]) = \alpha}{\Psi\ ;\ \mathcal{V} \vdash \alpha \equiv U[\sigma]}\ \text{v-n} \qquad \frac{\alpha \in \mathcal{V} \quad \text{unfold}_\Sigma(V[\theta]) = \alpha}{\Psi\ ;\ \mathcal{V} \vdash V[\theta] \equiv \alpha}\ \text{n-v}$$

$$\frac{\Psi\ ;\ \mathcal{V} \vdash \theta \equiv \sigma}{\Psi\ ;\ \mathcal{V} \vdash V[\theta] \equiv V[\sigma]}\ \text{refl} \qquad \frac{\Psi, \langle \mathcal{V}\ ;\ V[\theta] \equiv U[\sigma] \rangle\ ;\ \mathcal{V} \vdash \text{unfold}_\Sigma(V[\theta]) \equiv \text{unfold}_\Sigma(U[\sigma])}{\Psi\ ;\ \mathcal{V} \vdash V[\theta] \equiv U[\sigma]}\ \text{expd}$$

$$\frac{\begin{array}{c}(V \neq U) \quad \langle \mathcal{V}'\ ;\ V[\theta'] \equiv U[\sigma'] \rangle \in \Psi \\ \mathcal{V} \vdash \phi' : \mathcal{V}' \quad \Psi\ ;\ \mathcal{V} \Vdash \theta \equiv \phi' \circ \theta' \quad \Psi\ ;\ \mathcal{V} \Vdash \phi' \circ \sigma' \equiv \sigma\end{array}}{\Psi\ ;\ \mathcal{V} \vdash V[\theta] \equiv U[\sigma]}\ \text{def}$$

Fig. 2. Algorithmic rules for type equality.

The rules are shown in Figure 2. The judgment has the form $\Psi\ ;\ \mathcal{V} \vdash_\Sigma A \equiv B$, where $\mathcal{V}$ contains the free type variables in the types A and B, and Σ is a fixed *valid* signature containing type definitions of the form $V[\overline{\alpha}] \triangleq C$, and Ψ is a collection of *closures* $\langle \mathcal{V}\ ;\ V[\theta] \equiv U[\sigma] \rangle$. If a derivation can be constructed, all *closed instances* of all judgments $\Psi; \mathcal{V} \vdash V[\theta] \equiv U[\sigma]$ that occur are included in the resulting bisimulation (see the proof of Theorem 5.6). A closed instance of judgment $\Psi; \mathcal{V} \vdash V[\theta] \equiv U[\sigma]$ is obtained by applying a $\mathcal{V}$-closing substitution ϕ—that is, all types $\phi(V[\theta])$ and $\phi(U[\sigma])$ that have no free type variables. Recall that because type name definitions have no free variables, we treat the types $\phi(V[\theta])$ and $V[\phi \circ \theta]$ as syntactically equal. Last, because the signature Σ is fixed, we elide it from the rules in Figure 2.

In the type equality algorithm, the rules for type operators simply compare the components. If the type constructors (or the label sets in the $\oplus$ and $\&$ rules) do not match, then type equality fails, having constructed a counterexample to bisimulation. Similarly, two type variables are considered equal if and only if they have the same name, as in the v-v rule. We also include the v-n and n-v rules so that this comparison is made up to the unfolding of type name instantiations.

A rule of quasi-reflexivity is needed. In the algorithm of Gay and Hole, the rule would be merely reflexivity ($V \equiv V$) and is not needed. In our algorithm, the refl rule generalizes reflexivity by allowing identical type names to be instantiated by pointwise equal substitutions; without this rule, our algorithm would sometimes fail to recognize type names instantiated with equal types as equal.[4]

Now we come to the key rules, expd and def. In the expd rule, we unfold the definitions of $V[\theta]$ and $U[\sigma]$, and add the closure $\langle \mathcal{V}\ ;\ V[\theta] \equiv U[\sigma] \rangle$ to Ψ. (We do allow the expd rule to be applied when V and U are the same type name.) Since the equality of $V[\theta]$ and $U[\sigma]$ must hold for all of its closed instances, the extension of Ψ with the corresponding closure serves to remember exactly that.

The def rule only applies when there already exists a closure $\langle \mathcal{V}'\ ;\ V[\theta'] \equiv U[\sigma'] \rangle$ in Ψ with the same type names V and U as the goal $V[\theta] \equiv U[\sigma]$. In that case, we try to find a substitution

[4] This quasi-reflexivity rule also resembles a compatibility rule, but we reserve the term *compatability* for the idea that equal types can replace each other underneath any type operator, not just type name instantiations. A full-fledged compatability rule is not appropriate for our algorithm, so we retain the refl name for the rule we have.

ϕ' such that θ is equal to $\phi' \circ \theta'$ and $\phi' \circ \sigma'$ is equal to σ. The substitution ϕ' is computed by a standard matching algorithm on first-order terms (which is linear-time), applied on the syntactic structure of the types. Existence of such a substitution ensures that any closed instance of the goal $V[\theta] \equiv U[\sigma]$ is also a closed instance of $\langle \mathcal{V}' \ ; \ V[\theta'] \equiv U[\sigma'] \rangle$—those closed instances are already present in the (implicitly) constructed type bisimulation, so we can terminate our equality check, having successfully *detected a loop*.

The algorithm so far is sound but potentially non-terminating. There are two points of non-termination: (i) when encountering name/name equations, we can use the expd rule indefinitely, and (ii) we call the type equality judgment recursively in the def rule (by way of the equality judgment on substitutions). To ensure termination in the former case, we restrict the expd rule so that for any pair of type names V and U there is an upper bound on the number of closures of the form $\langle - \ ; \ V[-] \equiv U[-] \rangle$ allowed in Ψ. We define this upper bound as the *depth bound* of the algorithm and allow the programmer to specify this depth bound. Surprisingly, a depth bound of 1 suffices for all of our examples. This also removes the overlap between the expd and def rules.

In the latter case, instead of calling the general type equality algorithm, we introduce the notion of *rigid equality*, denoted by $\Psi; \mathcal{V} \Vdash A \equiv B$. The only difference between general equality and rigid equality is that we cannot employ the expd rule for rigid equality; otherwise, rigid equality has all of the other rules for equality but with both premises and conclusion using the rigid judgment. For instance, the rigid equality judgment includes a rigid analogue of the refl rule.

$$\frac{\Psi; \mathcal{V} \vdash \theta \equiv \sigma}{\Psi; \mathcal{V} \vdash V[\theta] \equiv V[\sigma]}\ \mathsf{refl} \qquad\qquad \frac{\Psi; \mathcal{V} \Vdash \theta \equiv \sigma}{\Psi; \mathcal{V} \Vdash V[\theta] \equiv V[\sigma]}\ \mathsf{r\text{-}refl}$$

Since the sizes of the types reduce in all equality rules except for expd, this algorithm now terminates. When comparing two instantiated type names, our algorithm first tries reflexivity, then tries to close a loop with def, and only if neither of these is applicable or fails do we expand the definitions with the expd rule. Gay and Hole's algorithm (with the small optimizations of reflexivity and internal renaming) is obtained as the specific instance of our algorithm when all type names have no parameters; this means our algorithm is both sound and complete on monomorphic types.

Both the general and rigid equality judgments are also extended to substitutions pointwise as the judgments $\Psi; \mathcal{V} \vdash \theta \equiv \sigma$ and $\Psi; \mathcal{V} \Vdash \theta \equiv \sigma$, respectively. The pointwise rules for equality of substitutions are as follows.

$$\frac{\Psi; \mathcal{V} \vdash \theta(\alpha) \equiv \sigma(\alpha) \quad (\forall \alpha \in \mathsf{dom}(\theta))}{\Psi; \mathcal{V} \vdash \theta \equiv \sigma}\ \mathsf{subs} \qquad\qquad \frac{\Psi; \mathcal{V} \Vdash \theta(\alpha) \equiv \sigma(\alpha) \quad (\forall \alpha \in \mathsf{dom}(\theta))}{\Psi; \mathcal{V} \Vdash \theta \equiv \sigma}\ \mathsf{r\text{-}subs}$$

5.1 Type Equality Declarations

In the following section, we will prove the soundness of the preceding algorithm. The algorithm, however, is unavoidably incomplete. One of the primary sources of incompleteness in our algorithm is its inability to *generalize the coinductive hypothesis*. As an illustration, consider the following two types D_0 and D'_0, which only differ in the names but have the same structure.

$$D[\kappa] \triangleq \oplus\{\mathbf{L} : D[D[\kappa]], \mathbf{R} : \kappa\} \qquad\qquad D_0 \triangleq \oplus\{\mathbf{L} : D[D_0], \$: \mathbf{1}\}$$

$$D'[\kappa] \triangleq \oplus\{\mathbf{L} : D'[D'[\kappa]], \mathbf{R} : \kappa\} \qquad\qquad D'_0 \triangleq \oplus\{\mathbf{L} : D'[D'_0], \$: \mathbf{1}\}$$

To establish $D_0 \equiv D'_0$, our algorithm explores the $\mathbf{L}$ branch and checks $D[D_0] \equiv D'[D'_0]$. A corresponding closure $\langle \cdot \ ; \ D[D_0] \equiv D'[D'_0] \rangle$ is added to Γ, and our algorithm then checks $D[D[D_0]] \equiv D'[D'[D'_0]]$. This process repeats until it exceeds the depth bound and terminates with an inconclusive answer. What the algorithm never realizes is that $D[\kappa] \equiv D'[\kappa]$ for *all* types κ; it fails to generalize to this hypothesis and is always inserting closures over closed types to Ψ.

To allow a recourse, we permit the programmer to declare (with concrete syntax)

```
eqtype D[k] = D'[k]
```

an equality constraint easily verified by our algorithm. We then *seed* the Ψ in the equality algorithm with the corresponding closure from the eqtype declaration, which can then be used to establish $D_0 \equiv D_0'$. In other words, we can derive

$$\cdot \,;\, \langle \kappa \,;\, D[\kappa] \equiv D'[\kappa] \rangle \vdash D_0 \equiv D_0' \,.$$

Upon exploring the **L** branch, the goal is reduced to

$$\cdot \,;\, \langle \kappa \,;\, D[\kappa] \equiv D'[\kappa] \rangle, \langle \cdot \,;\, D_0 \equiv D_0' \rangle \vdash D[D_0] \equiv D'[D_0'] \,.$$

As required by the def rule, our algorithm now looks for a substitution ϕ' such that $D_0/\kappa \equiv \phi' \circ \mathrm{id}_\kappa$ and $\phi' \circ \mathrm{id}_\kappa \equiv D_0'/\kappa$, reducing the problem to the question whether $D_0 \equiv D_0'$. Since the latter has already been collected as a declaration, we terminate our deduction concluding that $D_0 \equiv D_0'$.

In the implementation, we first collect all eqtype declarations in the program into a global set of closures Ψ_0. We then validate the eqtype declarations in a mutually coinductive way by checking $\Psi_0 \,;\, \mathcal{V} \vdash V[\theta] \equiv U[\sigma]$ for every closure $\langle \mathcal{V} \,;\, V[\theta] \equiv U[\sigma] \rangle \in \Psi_0$. Crucially, we insist that each of these checks begins by applying the expd rule; without this requirement, the algorithm with eqtype declarations would be unsound.

One final note on the refl rule: a type name may *not* actually depend on its parameter. As a simple example, we have $V[\alpha] \triangleq \mathbf{1}$; a more complicated one would be $V[\alpha] \triangleq \oplus\{a\colon V[V[\alpha]], b\colon \mathbf{1}\}$. When applying refl, we would like to conclude that $V[A] \equiv V[B]$ regardless of A and B. This could be easily established with an equality type declaration eqtype $V[\alpha] = V[\beta]$. To avoid this syntactic overhead for the programmer, we determine for each parameter α of each type name V whether its definition is non-variant in α. This information is recorded in the signature and used when applying the reflexivity rule by ignoring non-variant arguments.

However, nearly all of these non-variant arguments are introduced by internal renaming. In practice, programmers do not naturally write code that uses non-variant arguments: programmers subconsciously understand that non-variant arguments are irrelevant. Moreover, those non-variant arguments that are introduced by internal renaming could be avoided by altering the internal renaming to use a subset of, rather than all, variables when introducing an internal name. For this reason, as well as the fact that a formal development of non-variance would take us too far afield, we choose not to incorporate the ignoring of non-variant arguments in the following soundness proof.

5.2 Soundness

At a high level, our algorithm is sound if algorithmic equality of types implies semantic equality (i.e., bisimilarity) of those types. Roughly speaking, this means proving that the algorithmic equality $\cdot;\mathcal{V} \vdash V[\theta] \equiv U[\sigma]$ implies the semantic equality $\mathcal{V} \vDash V[\theta] \equiv U[\sigma]$, for all $V[\theta]$ and $U[\sigma]$.

Although this statement happens to be provable without generalization, it is not general enough to capture soundness of our full algorithm. Specifically, the empty set of closures in the algorithmic equality judgment $\cdot;\mathcal{V} \vdash V[\theta] \equiv U[\sigma]$ means that this statement does not account for the system of eqtype declarations described previously—it only describes soundness when no eqtype declarations are used.

Following the mutually coinductive nature of eqtype declarations, soundness of our full algorithm is stated with a *collection* of algorithmic equality derivations.

Theorem (Soundness). *Let Ψ_0 be a set of closures, $\{\langle \mathcal{V}_i\ ;\ V_i[\theta_i] \equiv U_i[\sigma_i]\rangle \mid i \in I\}$, indexed by a set I. If $(\mathcal{D}_i)_{i\in I}$ is a collection of derivations such that each $\mathcal{D}_i$ derives $\Psi_0; \mathcal{V}_i \vdash V_i[\theta_i] \equiv U_i[\sigma_i]$ and has the* expd *rule at its root, then $\mathcal{V}_i \vDash V_i[\theta_i] \equiv U_i[\sigma_i]$ for all $i \in I$.*

The shared set of closures, Ψ_0, serves to implicitly tie a mutually coinductive knot among the derivations. (The requirement that the expd rule occurs at each derivation's root is necessary to prevent pathological cases in which two or more identical eqtype declarations are false but able to mutually justify each other via the def rule, such as identical declarations eqtype $V[\mathbf{1}] \equiv V[\alpha]$ when V is defined as $V[\alpha] \triangleq \alpha$.)

Overall, our proof strategy for this theorem is to use bisimulation-up-to techniques, namely the up-to-reflexivity, up-to-transitivity, and up-to-context techniques described by Sangiorgi [46]. Let $\mathcal{R}$ be the relation that consists of all closed instances of all judgments $\Psi; \mathcal{V} \vdash V[\theta] \equiv U[\sigma]$ (or $\Psi; \mathcal{V} \Vdash V[\theta] \equiv U[\sigma]$) that appear in one of the derivations $(\mathcal{D}_i)_{i\in I}$—that is, exactly those pairs $(\phi(V[\theta]), \phi(U[\sigma]))$ for all $\mathcal{V}$-closing substitutions ϕ for all judgments $\Psi; \mathcal{V} \vdash V[\theta] \equiv U[\sigma]$ (or $\Psi; \mathcal{V} \Vdash V[\theta] \equiv U[\sigma]$) that appear in some derivation $\mathcal{D}_i$. We shall show that this relation $\mathcal{R}$, although not itself a bisimulation, is a bisimulation up to reflexivity, transitivity, and context. Sangiorgi's results will then allow us to deduce that the relation $\mathcal{R}$ is contained in bisimilarity; since $\mathcal{R}$ includes all closed instances of the root judgments $\Psi_0; \mathcal{V}_i \vdash V_i[\theta_i] \equiv U_i[\sigma_i]$ by construction, we will then be able to conclude that $\mathcal{V}_i \vDash V_i[\theta_i] \equiv U_i[\sigma_i]$ for all $i \in I$ and that our algorithm is indeed sound.

Using Sangiorgi's results, showing that the relation $\mathcal{R}$ is a bisimulation up to reflexivity, transitivity, and context amounts to proving that $\mathcal{R}$ progresses to its reflexive, transitive, and contextual closure, which we write as $\mathcal{F}(\mathcal{R})$. What exactly do we mean here by contextual closure? For our proof, a *context* will be any type of the form $V[- \circ \theta]$, where the $(-)$ denotes a hole into which a substitution may be placed. Because $\mathcal{F}(\mathcal{R})$ is closed under context, if $(\phi_1(\alpha), \phi_2(\alpha)) \in \mathcal{F}(\mathcal{R})$ for all α in θ's codomain, then $(V[\phi_1 \circ \theta], V[\phi_2 \circ \theta]) \in \mathcal{F}(\mathcal{R})$. (Notice that because $\mathcal{F}(\mathcal{R})$ only relates *closed* types, the images of substitutions ϕ_1 and ϕ_2 that fill the holes in contexts must never include types with free variables.)

(It should be noted that the hole $-$ that appears in the context $V[-\circ\theta]$ is actually, in some sense, the outermost operation to be applied. If we rewrite this, it is morally something like $-(\theta(V))$: first, the substitution θ is applied to V, then the substitution represented by the hole $-$ is applied to that type.)

Before proving the progression from $\mathcal{R}$ to $\mathcal{F}(\mathcal{R})$, it is convenient to prove several lemmas. First, we will prove a lemma about the unfoldings of type name instantiations under substitutions.

Lemma 5.1. *The types $\phi(V[\theta])$ and $\phi(\mathsf{unfold}_\Sigma(V[\theta]))$ have the same unfoldings.*

Proof. By induction on the structure of the type $V[\theta]$. We distinguish cases on the body of V's definition:

Case: Consider the case in which $V[\bar{\alpha}] \triangleq \oplus\{\ell\colon V_\ell[\theta_\ell]\}_{\ell\in L}$. In this case, $\phi(V[\theta])$ unfolds to $\oplus\{\ell\colon \phi(V_\ell[\theta \circ \theta_\ell])\}_{\ell\in L}$, and so is (the unfolding of) $\phi(\mathsf{unfold}_\Sigma(V[\theta]))$.

Cases: The cases for $\&$, $\otimes$, $\multimap$, and $\mathbf{1}$ are analogous.

Case: Consider the case in which $V[\overline{\alpha}] \triangleq \alpha$. In this case, $\phi(\mathsf{unfold}_\Sigma(V[\theta]))$ is, by definition, $\phi(\mathsf{unfold}_\Sigma(\theta(\alpha)))$. By internal renaming, $\theta(\alpha) = V'[\theta']$ for some $V'[\theta']$. Appealing to the inductive hypothesis at the smaller type $V'[\theta']$, we may deduce that $\phi(\theta(\alpha))$ and $\phi(\mathsf{unfold}_\Sigma(\theta(\alpha)))$, and hence $\phi(\mathsf{unfold}_\Sigma(V[\theta]))$, have the same unfoldings. Now observe that the unfolding of $\phi(V[\theta])$ is that of $V[\phi \circ \theta]$, which is that of $\phi(\theta(\alpha))$. Therefore, the types $\phi(V[\theta])$ and $\phi(\mathsf{unfold}_\Sigma(V[\theta]))$ have the same unfoldings. ☐

Next, we will prove as a lemma that algorithmic subderivations involving structural types behave consistently with the relation $\mathcal{R}$ as a bisimulation up to $\mathcal{F}$.

Lemma 5.2. *If a judgment of the form $\Psi; \mathcal{V} \vdash A \equiv B$ or $\Psi; \mathcal{V} \vdash \alpha \equiv U[\sigma]$ or $\Psi; \mathcal{V} \vdash V[\theta] \equiv \alpha$ appears in some derivation $\mathcal{D}_i$, then the closed instances of the compared types have matching actions and $\mathcal{F}(\mathcal{R})$-related continuation types.*

Proof. We distinguish cases on the rule that derives the given judgment within derivation $\mathcal{D}_i$:

Case: Consider the case in which the $\oplus$ rule is used to derive the $\Psi; \mathcal{V} \vdash A \equiv B$ judgment: (i) $A = \oplus\{\ell\colon V_\ell[\theta_\ell]\}_{\ell \in L}$; (ii) $B = \oplus\{\ell\colon U_\ell[\sigma_\ell]\}_{\ell \in L}$; and (iii) there exist subderivations of $\Psi; \mathcal{V} \vdash V_\ell[\theta_\ell] \equiv U_\ell[\sigma_\ell]$, for all $\ell \in L$, within $\mathcal{D}_i$. Notice that $\phi(A) = \oplus\{\ell\colon \phi(V_\ell[\theta_\ell])\}_{\ell \in L}$ and similarly for $\phi(B)$. These types indeed have matching actions. Their continuation types, $\phi(V_\ell[\theta_\ell])$ and $\phi(U_\ell[\sigma_\ell])$, are $\mathcal{R}$-related on the basis of the judgment $\Psi; \mathcal{V} \vdash V_\ell[\theta_\ell] \equiv U_\ell[\sigma_\ell]$ that appears in $\mathcal{D}_i$. The relation $\mathcal{F}(\mathcal{R})$ contains $\mathcal{R}$, so the continuation types are also $\mathcal{F}(\mathcal{R})$-related.

Cases: The cases for the $\&$, $\otimes$, $\multimap$, and $\mathbf{1}$ rules are analogous.

Cases: Consider the cases in which either the v-v, v-n, or n-v rule is used to derive the given judgment. In each of these cases, at least one of the compared types is a variable $\alpha \in \mathcal{V}$, and the other type has α as its unfolding. Thus, according to Lemma 5.1, it suffices to show that the types $\phi(\alpha)$ and $\phi(\alpha)$ have matching actions and $\mathcal{F}(\mathcal{R})$-related continuation types. Because those types are syntactically equal, they have matching actions and syntactically equal continuation types. The relation $\mathcal{F}(\mathcal{R})$ is closed reflexively, so these continuation types are also $\mathcal{F}(\mathcal{R})$-related. ◻

Next, we have a lemma that verifies that closures behave consistently with the relation $\mathcal{R}$ as a bisimulation up to $\mathcal{F}$.

Lemma 5.3. *If a closure $\langle \mathcal{V} \;;\; V[\theta] \equiv U[\sigma] \rangle$ appears in derivation $\mathcal{D}_i$, then for all $\mathcal{V}$-closing substitutions ϕ, the closed types $\phi(V[\theta])$ and $\phi(U[\sigma])$ have matching actions and $\mathcal{F}(\mathcal{R})$-related continuation types.*

Proof. Because the context of closures behaves monotonically within a derivation, there are two possibilities for the location at which the closure $\langle \mathcal{V} \;;\; V[\theta] \equiv U[\sigma] \rangle$ arose: either it arose from an instance of the expd rule within $\mathcal{D}_i$ or it was already present in the initial context Ψ_0:

Case: Consider the case in which $\langle \mathcal{V} \;;\; V[\theta] \equiv U[\sigma] \rangle$ arose from an instance of the expd rule within $\mathcal{D}_i$. In this case, a judgment $\Psi, \langle \mathcal{V} \;;\; V[\theta] \equiv U[\sigma] \rangle; \mathcal{V} \vdash \mathsf{unfold}_\Sigma(V[\theta]) \equiv \mathsf{unfold}_\Sigma(U[\sigma])$ appears in $\mathcal{D}_i$, for some set of closures Ψ. The types $\mathsf{unfold}_\Sigma(V[\theta])$ and $\mathsf{unfold}_\Sigma(U[\sigma])$ are, in general, open types that are either structural or variables. It therefore follows from Lemma 5.2 that the types $\phi(\mathsf{unfold}_\Sigma(V[\theta]))$ and $\phi(\mathsf{unfold}_\Sigma(U[\sigma]))$ have matching actions and $\mathcal{F}(\mathcal{R})$-related continuation types.

By Lemma 5.1, the unfoldings of $\phi(\mathsf{unfold}_\Sigma(V[\theta]))$ and $\phi(V[\theta])$ are equal; likewise, the unfoldings of $\phi(\mathsf{unfold}_\Sigma(U[\sigma]))$ and $\phi(U[\sigma])$ are equal. We may therefore conclude that the types $\phi(V[\theta])$ and $\phi(U[\sigma])$ also have matching actions and $\mathcal{F}(\mathcal{R})$-related continuation types, as required.

Case: Consider the case in which $\langle \mathcal{V} \;;\; V[\theta] \equiv U[\sigma] \rangle$ was already present in the initial context Ψ_0. Then there must exist $k \neq i$ such that $V[\theta] = V_k[\theta_k]$ and $U[\sigma] = U_k[\sigma_k]$ (in the syntactic sense). The derivation $\mathcal{D}_k$ derives $\Psi_0; \mathcal{V}_k \vdash V_k[\theta_k] \equiv U_k[\sigma_k]$, using the expd rule at its root. Therefore, we may appeal to the preceding reasoning for the expd rule to conclude that the types $\phi(V_k[\theta_k])$ and $\phi(U_k[\sigma_k])$ have matching actions and $\mathcal{F}(\mathcal{R})$-related continuation types. ◻

Now we can prove that the relation $\mathcal{R}$ is a bisimulation up to $\mathcal{F}$, the reflexive, transitive, and contextual closure—that is, that $\mathcal{R}$ progresses to $\mathcal{F}(\mathcal{R})$. This represents a major portion of, but not quite all, soundness.

Lemma 5.4.

- *If $\Psi; \mathcal{V} \vdash V[\theta] \equiv U[\sigma]$ appears in some derivation $\mathcal{D}_i$, then $\phi(V[\theta])$ and $\phi(U[\sigma])$ have matching actions and $\mathcal{F}(\mathcal{R})$-related continuation types, for all $\mathcal{V}$-closing substitutions ϕ.*
- *If $\Psi; \mathcal{V} \vdash \theta \equiv \sigma$ appears in some derivation $\mathcal{D}_i$, then $\phi(V[\theta])$ and $\phi(V[\sigma])$ have matching actions and $\mathcal{F}(\mathcal{R})$-related continuation types, for all type names V and $\mathcal{V}$-closing substitutions ϕ.*

Proof. By mutual induction on the structure of the derivations of $\Psi; \mathcal{V} \vdash \theta \equiv \sigma$ and $\Psi; \mathcal{V} \vdash V[\theta] \equiv U[\sigma]$, respectively.[5]

- To prove the first statement, we begin by distinguishing cases on the rule that was used to derive the judgment $\Psi; \mathcal{V} \vdash V[\theta] \equiv U[\sigma]$ within $\mathcal{D}_i$:

  Case: Consider the case in which the judgment was derived by the expd rule: there exists a subderivation of $\Psi; \mathcal{V}, \langle \mathcal{V} \ ; \ V[\theta] \equiv U[\sigma] \rangle \vdash \mathsf{unfold}_\Sigma(V[\theta]) \equiv \mathsf{unfold}_\Sigma(U[\sigma])$. By Lemma 5.3, the types $\phi(V[\theta])$ and $\phi(U[\sigma])$ have matching actions and $\mathcal{F}(\mathcal{R})$-related continuation types.

  Case: Consider the case in which the judgment was derived by the def rule: in this case, there exist (i) a closure $\langle \mathcal{V}' \ ; \ V[\theta'] \equiv U[\sigma'] \rangle \in \Psi$; and subderivations of both (ii) $\Psi; \mathcal{V} \Vdash \theta \equiv \phi' \circ \theta'$ and (iii) $\Psi; \mathcal{V} \Vdash \phi' \circ \sigma' \equiv \sigma$ for some substitution $\mathcal{V} \vdash \phi' : \mathcal{V}'$.

  By appealing to the inductive hypothesis on the subderivations of $\Psi; \mathcal{V} \vdash \theta \equiv \phi' \circ \theta'$ and $\Psi; \mathcal{V} \vdash \phi' \circ \sigma' \equiv \sigma$, we may deduce that $\phi(V[\theta])$ and $\phi(V[\phi' \circ \theta'])$ have matching actions and $\mathcal{F}(\mathcal{R})$-related continuation types and, likewise, that $\phi(U[\phi' \circ \sigma'])$ and $\phi(U[\sigma])$ have matching actions and $\mathcal{F}(\mathcal{R})$-related continuation types. Last, by Lemma 5.3 on the closure $\langle \mathcal{V}' \ ; \ V[\theta'] \equiv U[\sigma'] \rangle$, the closed types $(\phi \circ \phi')(V[\theta'])$ and $(\phi \circ \phi')(U[\sigma'])$ also have matching actions and $\mathcal{F}(\mathcal{R})$-related continuation types.

  Notice that $\phi(V[\phi' \circ \theta'])$ and $(\phi \circ \phi')(V[\theta'])$ are syntactically equal types; similarly, $(\phi \circ \phi')(U[\sigma'])$ and $\phi(U[\phi' \circ \sigma'])$ are syntactically equal. Transitively, $\phi(V[\theta])$ and $\phi(U[\sigma])$ have matching actions. Moreover, because the relation $\mathcal{F}(\mathcal{R})$ is closed transitively, they also have $\mathcal{F}(\mathcal{R})$-related continuation types: from the preceding, we know that the continuation types of $\phi(V[\theta])$ are related to those of $\phi(V[\phi' \circ \theta'])$, which are in turn related to the continuation types of $\phi(U[\phi' \circ \sigma'])$ and thereby related to the continuation types of $\phi(U[\sigma])$.

  Case: Consider the case in which the judgment was derived by the refl rule: (i) the type names V and U are identical and (ii) there exists a subderivation of $\Psi; \mathcal{V} \vdash \theta \equiv \sigma$. Appealing to the inductive hypothesis on this subderivation, we may conclude that $\Psi; \mathcal{V} \vdash V[\theta] \equiv V[\sigma]$, as required.

- To prove the second part, we are given a derivation of $\Psi; \mathcal{V} \vdash \theta \equiv \sigma$ and consider an arbitrary type name $V \in \mathrm{dom}(\Sigma)$; we must show, for all $\mathcal{V}$-closing substitutions ϕ, that $\phi(V[\theta])$ and $\phi(V[\sigma])$ have matching actions and $\mathcal{F}(\mathcal{R})$-related continuation types. We begin by distinguishing cases on the body of V's definition, $V[\overline{\alpha}] \triangleq A$, in the signature Σ:

  Case: Consider the case in which $V[\overline{\alpha}] \triangleq \oplus\{\ell : V_\ell[\theta_\ell]\}_{\ell \in L}$. In this case, the unfoldings of $\phi(V[\theta])$ and $\phi(V[\sigma])$ are $\oplus\{\ell : (\phi \circ \theta)(V_\ell[\theta_\ell])\}_{\ell \in L}$ and $\oplus\{\ell : (\phi \circ \sigma)(V_\ell[\theta_\ell])\}_{\ell \in L}$, respectively.

[5] At first thought, it might be surprising that this proof uses an inductive argument. However, the induction is needed to drill down through type name definitions of the form $V[\overline{\beta}] \triangleq \alpha$ to the point at which a structural type appears.

These types indeed have matching actions, namely $\oplus\ell$ for all $\ell \in L$, with continuation types $(\phi \circ \theta)(V_\ell[\theta_\ell])$ and $(\phi \circ \sigma)(V_\ell[\theta_\ell])$, respectively. Notice that these continuation types are syntactically equal to $V_\ell[(\phi \circ \theta) \circ \theta_\ell]$ and $V_\ell[(\phi \circ \sigma) \circ \theta_\ell]$, respectively.

By inversion on the given derivation of $\Psi; \mathcal{V} \vdash \theta \equiv \sigma$, we may deduce that for all $\alpha \in \mathrm{dom}(\theta)$, the judgment $\Psi; \mathcal{V} \vdash \theta(\alpha) \equiv \sigma(\alpha)$ appears in derivation $\mathcal{D}_i$. Consequently, the closed types $(\phi \circ \theta)(\alpha)$ and $(\phi \circ \sigma)(\alpha)$ are $\mathcal{R}$-related, for all $\alpha \in \mathrm{dom}(\theta)$. Because the construction $\mathcal{F}(\mathcal{R})$ is closed under a set of faithful contexts that includes the context $V_\ell[- \circ \theta_\ell]$, the types $V_\ell[(\phi \circ \theta) \circ \theta_\ell]$ and $V_\ell[(\phi \circ \sigma) \circ \theta_\ell]$ are $\mathcal{F}(\mathcal{R})$-related. It follows that $\phi(V[\theta])$ and $\phi(V[\sigma])$ have $\mathcal{F}(\mathcal{R})$-related continuation types, as required.

Cases: The cases for definitions that use structural types $\&$, $\otimes$, $\multimap$, and $\mathbf{1}$ are analogous.

Case: Consider the case in which $V[\overline{\alpha}] \triangleq \alpha$; as usual, we must show that $\phi(V[\theta])$ and $\phi(V[\sigma])$ have matching actions and $\mathcal{F}(\mathcal{R})$-related continuation types. By Lemma 5.1, it suffices to show that $\phi(\theta(\alpha))$ and $\phi(\sigma(\alpha))$ have matching actions and $\mathcal{F}(\mathcal{R})$-related continuation types. After internal renaming, each substitution's image contains only variables and type name instantiations—never structural types. Thus, we may distinguish four subcases on the shapes of types $\theta(\alpha)$ and $\sigma(\alpha)$:

Subcases: Consider the subcases in which at least one of the types $\theta(\alpha)$ and $\sigma(\alpha)$ is a variable $\beta \in \mathcal{V}$. By inversion on the given derivation of $\Psi; \mathcal{V} \vdash \theta \equiv \sigma$, we may deduce that there exists a derivation of $\Psi; \mathcal{V} \vdash \theta(\alpha) \equiv \sigma(\alpha)$ within $\mathcal{D}_i$. On this basis, we appeal to Lemma 5.2, which covers all cases in which at least one of the compared types is not a type name instantiation, to conclude that $\phi(\theta(\alpha))$ and $\phi(\sigma(\alpha))$ indeed have matching actions and $\mathcal{F}(\mathcal{R})$-related continuation types.

Subcase: Consider the subcase in which both $\theta(\alpha)$ and $\sigma(\alpha)$ are type name instantiations $V'[\theta']$ and $U'[\sigma']$, respectively. Once again, by inversion on the given derivation of $\Psi; \mathcal{V} \vdash \theta \equiv \sigma$, we may deduce that there exists a subderivation of $\Psi; \mathcal{V} \vdash \theta(\alpha) \equiv \sigma(\alpha)$ within $\mathcal{D}_i$. Because both $\theta(\alpha)$ and $\sigma(\alpha)$ are type name instantiations, we may appeal to the inductive hypothesis on this subderivation. Thus, we deduce that $\phi(\theta(\alpha))$ and $\phi(\sigma(\alpha))$ have matching actions and $\mathcal{F}(\mathcal{R})$-related continuation types. □

Now we are nearly ready to apply the results of Sangiorgi [46] to conclude that our algorithm is sound. Before doing so, however, we need one final lemma: we must prove that the set of contexts $C[-]$ of the form $V[- \circ \sigma]$ constitute a *faithful* set of contexts, as defined by Sangiorgi. As Sangiorgi demonstrates, the up-to-context technique for bisimulation is unsound unless the considered contexts are faithful, so we must prove such a lemma.

In our setting, a sufficient condition for a set of contexts $C[-]$ to be faithful is the following. For each context $C[-]$ in the set, whenever the filled context $C[\theta]$ has an action with some continuation type, then one of the following conditions must hold: either

- (i) the action of $C[\theta]$ arises from the context $C[-]$ alone, and there exists a context $C'[-]$ in the set such that (ii) the continuation type has the form $C'[\theta]$ and (iii) this, including the choice of $C'[-]$, happens parametrically in θ; **or**
- (i) the action of $C[\theta]$ arises from an action of $\theta(\alpha)$, for some $\alpha \in \mathrm{dom}(\theta)$; (ii) the continuation type has the form $C[\theta']$, for some θ' such that $\theta'(\alpha)$ is the continuation type of $\theta(\alpha)$'s action; and (iii) this happens parametrically in θ, depending only on the action taken by $\theta(\alpha)$.

Recall that faithful contexts and the up-to-context technique are only sensible when applied to substitutions whose images contain only closed types.

We now prove faithfulness.

Lemma 5.5. *The contexts $C[-]$ of the form $V[- \circ \sigma]$ are faithful.*

Proof. To prove that the set of contexts of the form $C[-] = V[- \circ \sigma]$ is faithful, we choose an arbitrary such context and use structural induction on the substitution σ that appears as part of this context, distinguishing cases on the structure of V's definition within the signature Σ:

Case: Consider the case in which $V[\overline{\alpha}] \triangleq \oplus\{\ell\colon V_\ell[\theta_\ell]\}_{\ell \in L}$. Here, $C[\theta] = V[\theta \circ \sigma]$ unfolds to $\oplus\{\ell\colon V_\ell[\theta \circ (\sigma \circ \theta_\ell)]\}_{\ell \in L}$. The filled context $C[\theta]$ has actions $\oplus k$, for all $k \in L$, with continuation types $V_k[\theta \circ (\sigma \circ \theta_k)]$. For action $\oplus k$, we may choose $C'[-] = V_k[- \circ (\sigma \circ \theta_k)]$, observing that $C[\theta]$'s continuation type under this action is $C'[\theta]$. Moreover, all of this reasoning is parametric in θ. Therefore, the context $C[-] = V[- \circ \sigma]$ is faithful when $V[\overline{\alpha}]$ is defined to be an internal choice.

Cases: The cases for $\&$, $\otimes$, and $\multimap$ are analogous.

Case: Consider the case in which $V[\overline{\alpha}] \triangleq \mathbf{1}$. In this case, the only action offered by $C[\theta]$ is the action $\mathbf{1}$ that has continuation "type" ϵ. This is not actually a type but rather a technical device that would be needed in an explicit labeled transition system for types. If we also include $C'[-] = \epsilon$ as a (constant) context in our set of contexts, then the continuation ϵ can be expressed as $C'[\theta]$. Using the constant context $C'[-] = \epsilon$ imposes on us the obligation to prove that $C'[-] = \epsilon$ satisfies the preceding condition. Indeed, the condition is trivially satisfied because $C'[-] = \epsilon$ never offers actions.

Case: Consider the case in which $V[\overline{\alpha}] \triangleq \alpha$. In this case, the unfolding of $C[\theta] = V[\theta \circ \sigma]$ is the unfolding of $(\theta \circ \sigma)(\alpha)$. We distinguish cases on the shape of $\sigma(\alpha)$. Recall that after internal renaming, the image of a substitution never contains structural types. We have the following subcases:

- **Subcase:** Consider the subcase in which $\sigma(\alpha) = \beta$ for some $\beta \in \mathrm{dom}(\theta)$. In this subcase, the unfolding of $C[\theta] = V[\theta \circ \sigma]$ is the unfolding of $\theta(\beta)$, so they have the same actions and the same continuation types. After internal renaming, continuation types are always syntactically type name instantiations. (They cannot be variables because the notion of progression, and therefore continuation type, is defined only for closed types.) Therefore, we can always construct an internally renamed substitution $\cdot \vdash \theta' : \mathrm{dom}(\theta)$ such that $\theta(\beta)$'s continuation type is $\theta'(\beta)$. Observe that the unfolding of $C[\theta'] = V[\theta' \circ \sigma]$ is the unfolding of $\theta'(\beta)$. In other words, $C[\theta]$'s continuation type indeed has the form $C[\theta']$. Moreover, this happens parametrically in θ, depending only on the action taken by $\theta(\beta)$.
- **Subcase:** Consider the subcase in which $\sigma(\alpha) = U[\sigma']$, for some $U[\sigma']$. In this subcase, the unfolding of $(\theta \circ \sigma)(\alpha)$ is the unfolding of $U[\theta \circ \sigma']$. Because the substitution σ' occurs as a proper subterm of σ, we may appeal to the inductive hypothesis to deduce that the context $U[- \circ \sigma']$ is faithful. Because $C[\theta] = V[\theta \circ \sigma]$ unfolds to the unfolding of $U[\theta \circ \sigma']$ parametrically in θ, it follows that $C[-] = V[- \circ \sigma]$ is also faithful. □

Theorem 5.6 (Soundness). *Let Ψ_0 be a set of closures, $\{\langle \mathcal{V}_i \ ;\ V_i[\theta_i] \equiv U_i[\sigma_i]\rangle \mid i \in I\}$, indexed by a set I. If $(\mathcal{D}_i)_{i \in I}$ is a collection of derivations such that each $\mathcal{D}_i$ derives $\Psi_0; \mathcal{V}_i \vdash V_i[\theta_i] \equiv U_i[\sigma_i]$ and has the* expd *rule at its root, then $\mathcal{V}_i \vDash V_i[\theta_i] \equiv U_i[\sigma_i]$ for all $i \in I$.*

Proof. The preceding lemma establishes that contexts of the form $V[- \circ \theta_0]$ are faithful. Then, according to Sangiorgi, the relation $\mathcal{R}$ is contained in bisimilarity if $\mathcal{R}$ progresses to $\mathcal{F}(\mathcal{R})$, its reflexive, transitive, and contextual closure. This is proved by Lemma 5.4, so $\mathcal{R}$ indeed contained in bisimilarity. In particular, because $\mathcal{R}$ includes the pairs of type name instantiations at the root of each derivation $\mathcal{D}_i$, we conclude that $\mathcal{V}_i \vDash V_i[\theta_i] \equiv U_i[\sigma_i]$ for all $i \in I$. □

Table 2. Session Types with Operational Description

Type	Cont.	Process Term	Cont.	Description
$c : \oplus\{\ell : A_\ell\}_{\ell \in L}$	$c : A_k$	$c.k \; ; \; P$	P	Send label k on c
		$\mathsf{case}\; c \; (\ell \Rightarrow Q_\ell)_{\ell \in L}$	Q_k	Receive label k on c
$c : \&\{\ell : A_\ell\}_{\ell \in L}$	$c : A_k$	$\mathsf{case}\; c \; (\ell \Rightarrow P_\ell)_{\ell \in L}$	P_k	Receive label k on c
		$c.k \; ; \; Q$	Q	Send label k on c
$c : A \otimes B$	$c : B$	$\mathsf{send}\; c \; w \; ; \; P$	P	Send channel $w : A$ on c
		$y \leftarrow \mathsf{recv}\; c \; ; \; Q_y$	$Q_y[w/y]$	Receive channel $w : A$ on c
$c : A \multimap B$	$c : B$	$y \leftarrow \mathsf{recv}\; c \; ; \; P_y$	$P_y[w/y]$	Receive channel $w : A$ on c
		$\mathsf{send}\; c \; w \; ; \; Q$	Q	Send channel $w : A$ on c
$c : \mathbf{1}$	—	$\mathsf{close}\; c$	—	Send *close* on c
		$\mathsf{wait}\; c \; ; \; Q$	Q	Receive *close* on c

6 FORMAL LANGUAGE DESCRIPTION

In this section, we present the program constructs we have designed to realize nested polymorphism that have also been integrated with the Rast language [18–20] to support general-purpose programming. The underlying base system of session types is derived from a Curry-Howard interpretation [7, 8] of intuitionistic linear logic [27]. The key idea is that an intuitionistic linear sequent $A_1 \; A_2 \; \ldots \; A_n \vdash A$ is interpreted as the interface to a process P. We label each of the antecedents with a channel name x_i and the succedent with channel name x. The x_i's are *channels used by* P and x is the *channel provided by* P.

$$(x_1 : A_1) \; (x_2 : A_2) \ldots (x_n : A_n) \vdash P :: (x : A)$$

The resulting judgment formally states that process P provides a service of session type A along channel x while using the services of session types $A_1, \ldots, A_n$ provided along channels $x_1, \ldots, x_n$, respectively. All of these channels must be distinct. We abbreviate the antecedent of the sequent by Δ.

Due to the presence of type variables, the typing judgment is extended with $\mathcal{V}$ and written as

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

where $\mathcal{V}$ stores the type variables α, Δ represents the linear antecedents $x_i : A_i$, P is the process expression, and $x : A$ is the linear succedent. We maintain that all free type variables in Δ, P, and A are contained in $\mathcal{V}$. Finally, Σ is a fixed valid signature containing type and process definitions. Table 2 overviews the session types, their associated process terms, their continuation (both in types and terms), and operational description. For each type, the first line of that type's row in Table 2 describes the provider's viewpoint, whereas the second line describes the client's matching but dual viewpoint.

We formalize the operational semantics as a *multiset rewriting system* [9]. We introduce semantic objects $\mathsf{proc}(c, P)$ and $\mathsf{msg}(c, M)$ for processes P and messages M providing along channel c.

Definition 6.1 (Configuration). At runtime, a program is represented using a multiset of semantic objects denoting processes ($\mathsf{proc}(c, P)$) and messages ($\mathsf{msg}(c, M)$) defined as a *configuration*.

$$\mathcal{S} \; ::= \; \cdot \mid \mathcal{S}, \mathcal{S}' \mid \mathsf{proc}(c, P) \mid \mathsf{msg}(c, M)$$

We stipulate that no two distinct semantic objects in a configuration provide the same channel.

$$\begin{aligned}\text{Procs } P, Q &::= x.k\ ;\ P \mid \mathsf{case}\ x\ (\ell \Rightarrow Q_\ell)_{\ell \in L} \mid \mathsf{send}\ x\ w\ ;\ P \mid y \leftarrow \mathsf{recv}\ x\ ;\ Q \mid \mathsf{close}\ x \mid \mathsf{wait}\ x\ ;\ Q \\ &\mid\ x \leftrightarrow y \mid x \leftarrow f[\theta]\ \overline{y}\ ;\ P \\ \text{Messages } M &::= x.k\ ;\ x \leftrightarrow x' \mid x.k\ ;\ x' \leftrightarrow x \mid \mathsf{send}\ x\ w\ ;\ x \leftrightarrow x' \mid \mathsf{send}\ x\ w\ ;\ x' \leftrightarrow x \mid \mathsf{close}\ x\end{aligned}$$

Fig. 3. Grammar for processes and messages.

6.1 Statics and Semantics

We briefly review the structural types already existing in the Rast language. For reference, Figure 3 provides a grammar for process and message expressions. Inspired by sequent calculus, the type system is presented via left (L) and right (R) rules for each session type connective, depending on whether the connective appears to the left or right of the sequent. With our equirecursive interpretation of types, the occurrence of the connectives may result from an implicit unfolding of type names. The reduction rules in the operational semantics are presented via S rules where the sender creates a message and C rules where the receiver obtains the previously created message (S stands for send, and C stands for compute). The preceding convention is followed irrespective of which process is the provider or client of a channel. A final remark regarding messages: they can be typed exactly as processes and do not need any explicit rules!

The *internal choice* type constructor $\oplus\{\ell : A_\ell\}_{\ell \in L}$ is an n-ary labeled generalization of the additive disjunction $A \oplus B$. Operationally, it requires the provider of $x : \oplus\{\ell : A_\ell\}_{\ell \in L}$ to send a label $k \in L$ on channel x and continue to provide type A_k. The corresponding process term is written as $(x.k\ ;\ P)$, where the continuation P provides type $x : A_k$. Dually, the client must branch based on the label received on x using the process term $\mathsf{case}\ x\ (\ell \Rightarrow Q_\ell)_{\ell \in L}$, where Q_ℓ is the continuation in the ℓ-th branch.

$$\frac{(k \in L) \quad \mathcal{V}\ ;\ \Delta \vdash P :: (x : A_k)}{\mathcal{V}\ ;\ \Delta \vdash (x.k\ ;\ P) :: (x : \oplus\{\ell : A_\ell\}_{\ell \in L})}\ \oplus R$$

$$\frac{(\forall \ell \in L) \quad \mathcal{V}\ ;\ \Delta, (x : A_\ell) \vdash Q_\ell :: (z : C)}{\mathcal{V}\ ;\ \Delta, (x : \oplus\{\ell : A_\ell\}_{\ell \in L}) \vdash \mathsf{case}\ x\ (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (z : C)}\ \oplus L$$

Communication is asynchronous, so the client $(c.k\ ;\ Q)$ sends a message k along c and continues as Q without waiting for it to be received. As a technical device to ensure that consecutive messages on a channel arrive in order, the sender also creates a fresh continuation channel c' so that the message k is actually represented as $(c.k\ ;\ c \leftrightarrow c')$ (read: send k along c and continue along c'). When the message k is received along c, we select branch k and also substitute the continuation channel c' for c.

$$\begin{aligned}&(\oplus S) : \mathsf{proc}(c, c.k\ ;\ P)\ \mapsto\ \mathsf{proc}(c', P[c'/c]), \mathsf{msg}(c, c.k\ ;\ c \leftrightarrow c') \\ &(\oplus C) : \mathsf{msg}(c, c.k\ ;\ c \leftrightarrow c'), \mathsf{proc}(d, \mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell \in L})\ \mapsto\ \mathsf{proc}(d, Q_k[c'/c])\end{aligned}$$

The *external choice* constructor $\&\{\ell : A_\ell\}_{\ell \in L}$ generalizes additive conjunction and is the *dual* of internal choice reversing the role of the provider and client. Thus, the provider branches on the label $k \in L$ sent by the client.

$$\frac{(\forall \ell \in L) \quad \mathcal{V}\ ;\ \Delta \vdash P_\ell :: (x : A_\ell)}{\mathcal{V}\ ;\ \Delta \vdash \mathsf{case}\ x\ (\ell \Rightarrow P_\ell)_{\ell \in L} :: (x : \&\{\ell : A_\ell\}_{\ell \in L})}\ \& R$$

$$\frac{(k \in L) \quad \mathcal{V}\ ;\ \Delta, (x : A_k) \vdash Q :: (z : C)}{\mathcal{V}\ ;\ \Delta, (x : \&\{\ell : A_\ell\}_{\ell \in L}) \vdash (x.k\ ;\ Q) :: (z : C)}\ \& L$$

Rules $\& S$ and $\& C$ that follow describe the operational behavior of the provider and client respectively (c' fresh).

$(\&S) : \mathsf{proc}(d, c.k\ ;\ Q)\ \mapsto\ \mathsf{msg}(c', c.k\ ;\ c' \leftrightarrow c), \mathsf{proc}(d, Q[c'/c])$
$(\&C) : \mathsf{proc}(c, \mathsf{case}\ c\ (\ell \Rightarrow P_\ell)_{\ell \in L}), \mathsf{msg}(c', c.k\ ;\ c' \leftrightarrow c)\ \mapsto\ \mathsf{proc}(c', P_k[c'/c])$

The *tensor* operator $A \otimes B$ prescribes that the provider of $x : A \otimes B$ sends a channel, say w of type A and continues to provide type B. The corresponding process term is $\mathsf{send}\ x\ w\ ;\ P$, where P is the continuation. Correspondingly, its client must receive a channel on x using the term $y \leftarrow \mathsf{recv}\ x\ ;\ Q$, binding it to variable y and continuing to execute Q.

$$\frac{\cdot\ ;\ \mathcal{V} \vdash A \equiv A' \qquad \mathcal{V}\ ;\ \Delta \vdash P :: (x : B)}{\mathcal{V}\ ;\ \Delta, (y : A) \vdash (\mathsf{send}\ x\ y\ ;\ P) :: (x : A' \otimes B)}\ \otimes R$$

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

Linearity of channels is enforced by removing $y : A$ from the context Δ after it is sent. The type checker also uses the equality algorithm to confirm that the types A and A' match. Operationally, the provider $(\mathsf{send}\ c\ d\ ;\ P)$ sends the channel d and the continuation channel c' along c as a message and continues with executing P. The client receives channel d and continuation channel c' appropriately substituting them.

$(\otimes S) : \mathsf{proc}(c, \mathsf{send}\ c\ d\ ;\ P)\ \mapsto \mathsf{proc}(c', P[c'/c]), \mathsf{msg}(c, \mathsf{send}\ c\ d\ ;\ c \leftrightarrow c')$
$(\otimes C) : \mathsf{msg}(c, \mathsf{send}\ c\ d\ ;\ c \leftrightarrow c'), \mathsf{proc}(e, x \leftarrow \mathsf{recv}\ c\ ;\ Q)\ \mapsto\ \mathsf{proc}(e, Q[c', d/c, x])$

The dual operator $A \multimap B$ allows the provider to receive a channel of type A and continue to provide type B. The client of $A \multimap B$, however, sends the channel of type A and continues to use B using dual process terms as $\otimes$.

$$\frac{\mathcal{V}\ ;\ \Delta, (y : A) \vdash P :: (x : B)}{\mathcal{V}\ ;\ \Delta \vdash (y \leftarrow \mathsf{recv}\ x\ ;\ P) :: (x : A \multimap B)}\ \multimap R$$

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

$(\multimap S) : \mathsf{proc}(e, \mathsf{send}\ c\ d\ ;\ Q)\ \mapsto \mathsf{msg}(c', \mathsf{send}\ c\ d\ ;\ c' \leftrightarrow c), \mathsf{proc}(e, Q[c'/c])$
$(\multimap C) : \mathsf{proc}(c, x \leftarrow \mathsf{recv}\ c\ ;\ P), \mathsf{msg}(c', \mathsf{send}\ c\ d\ ;\ c' \leftrightarrow c)\ \mapsto\ \mathsf{proc}(c', P[c', d/c, x])$

The type $\mathbf{1}$ indicates *termination* requiring that the provider of $x : \mathbf{1}$ sends a *close* message, formally written as $\mathsf{close}\ x$ followed by terminating the communication. Correspondingly, the client of $x : \mathbf{1}$ uses the term $\mathsf{wait}\ x\ ;\ Q$ to wait for x to terminate before continuing with executing Q. Linearity enforces that the provider does not use any channels.

$$\frac{}{\mathcal{V}\ ;\ \cdot \vdash (\mathsf{close}\ x) :: (x : \mathbf{1})}\ \mathbf{1}R \qquad \frac{\mathcal{V}\ ;\ \Delta \vdash Q :: (z : C)}{\mathcal{V}\ ;\ \Delta, (x : \mathbf{1}) \vdash (\mathsf{wait}\ x\ ;\ Q) :: (z : C)}\ \mathbf{1}L$$

Operationally, the provider waits for the closing message, which has no continuation channel since the provider terminates.

$(\mathbf{1}S) : \mathsf{proc}(c, \mathsf{close}\ c)\ \mapsto\ \mathsf{msg}(c, \mathsf{close}\ c)$
$(\mathbf{1}C) : \mathsf{msg}(c, \mathsf{close}\ c), \mathsf{proc}(d, \mathsf{wait}\ c\ ;\ Q)\ \mapsto\ \mathsf{proc}(d, Q)$

A forwarding process $x \leftrightarrow y$ identifies the channels x and y so that any further communication along either x or y will be along the unified channel. Its typing rule corresponds to the logical rule of identity.

$$\frac{\cdot\ ;\ \mathcal{V} \vdash A \equiv A'}{\mathcal{V}\ ;\ y : A \vdash (x \leftrightarrow y) :: (x : A')}\ \mathsf{id}$$

Operationally, a process $c \leftrightarrow d$ *forwards* any message M that arrives on d to c and vice versa. Since channels are used linearly, the forwarding process can then terminate, ensuring proper renaming, as exemplified in the following rules.

$$(\mathrm{id}^+C) : \mathrm{msg}(d, M), \mathrm{proc}(c, c \leftrightarrow d) \ \mapsto \ \mathrm{msg}(c, M[c/d])$$
$$(\mathrm{id}^-C) : \mathrm{proc}(c, c \leftrightarrow d), \mathrm{msg}(e, M(c)) \ \mapsto \ \mathrm{msg}(e, M(c)[d/c])$$

We write $M(c)$ to indicate that c must occur in message M ensuring that M is the sole client of c. Since the forwarding process uses channel d and provides channel c, which in turn is used by the message provided on channel e, linearity enforces an ordering of sort $d < c < e$, thereby guaranteeing that d and e are distinct channels.

6.1.1 Process Definitions. Process definitions have the form $\Delta \vdash f[\overline{\alpha}] = P :: (x : A)$, where f is the name of the process and P its definition, with Δ being the channels used by f and $x : A$ being the offered channel. In addition, $\overline{\alpha}$ is a sequence of type variables that Δ, P, and A can refer to. These type variables are implicitly universally quantified at the outermost level and represent prenex polymorphism. All definitions are collected in the fixed global signature Σ. For a *valid signature*, we require that $\overline{\alpha} \ ; \ \Delta \vdash P :: (x : A)$ for every definition, thereby allowing definitions to be mutually recursive. A new instance of a defined process f can be spawned with the expression $x \leftarrow f[\theta] \ \overline{y} \ ; \ Q$, where $\overline{y}$ is a sequence of channels matching the antecedents Δ and θ is a substitution for the type variables $\overline{\alpha}$. The newly spawned process will use all variables in $\overline{y}$ and provide x to the continuation Q.

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

The declaration of f is looked up in the signature Σ (first premise), and subsitution θ is applied to the types of $\overline{y}$ (second premise). Similarly, the freshly created channel x has type B from the signature, under substitution θ. The type system additionally calls on to the equality check to verify that the types $\overline{\theta(B')}$ are pointwise equivalent to Δ' (equality judgment extended pointwise in the second premise in the def rule). The corresponding semantics rule also performs a similar substitution (a fresh).

$$(\mathrm{def}C) : \mathrm{proc}(c, x \leftarrow f[\theta] \ \overline{d} \ ; \ Q) \ \mapsto \ \mathrm{proc}(a, P_f[a/x, \overline{d}/\overline{y'}, \theta]), \ \mathrm{proc}(c, Q[a/x]),$$

where $\overline{y' : B'} \vdash f = P_f :: (x' : B) \in \Sigma$.

Sometimes a process invocation is a tail call, written without a continuation as $x \leftarrow f[\theta] \ \overline{y}$. This is shorthand for $x' \leftarrow f[\theta] \ \overline{y} \ ; \ x \leftrightarrow x'$ for a fresh variable x'—that is, we create a fresh channel and immediately identify it with x.

6.2 Type Safety

The extension of session types with nested polymorphism is proved type safe by the standard theorems of *preservation* and *progress*, also known as *session fidelity* and *deadlock freedom*.

6.2.1 Type Preservation. The key to preservation is defining the rules to *type a configuration*. We define a well-typed configuration using the judgment $\Delta_1 \vDash^{\Sigma}_{C} \mathcal{S} :: \Delta_2$ denoting that configuration $\mathcal{S}$ (recall Definition 6.1 of a configuration) uses channels Δ_1 and provides channels Δ_2.[6] A

[6] Although we use the same turnstile, $\vDash$, for semantic equality (Definitions 4.4 and 4.5) as here for configuration typing, the significantly different right-hand sides allow the two judgments to be easily distinguished.

$$\frac{}{\Delta \vDash_C (\cdot) :: \Delta}\ \text{emp} \qquad \frac{\Delta_1 \vDash_C \mathcal{S}_1 :: \Delta_2 \qquad \Delta_2 \vDash_C \mathcal{S}_2 :: \Delta_3}{\Delta_1 \vDash_C (\mathcal{S}_1, \mathcal{S}_2) :: \Delta_3}\ \text{comp} \qquad \frac{\cdot\ ;\ \Delta \vdash P :: (x : A)}{\Delta \vDash_C \mathsf{proc}(x, P) :: (x : A)}\ \text{proc}$$

$$\frac{\cdot\ ;\ \Delta \vdash M :: (x : A)}{\Delta \vDash_C \mathsf{msg}(x, M) :: (x : A)}\ \text{msg}$$

Fig. 4. Typing rules for a configuration.

configuration is always typed with respect to a valid signature Σ. Since the signature Σ is fixed, we elide it from the presentation.

The rules for typing a configuration are defined in Figure 4. The emp rule states that an empty configuration does not consume any channels but provides all channels it uses. The comp rule composes two configurations $\mathcal{S}_1$ and $\mathcal{S}_2$; $\mathcal{S}_1$ provides channels Δ_2, whereas $\mathcal{S}_2$ uses channels Δ_2. The rule proc creates a singleton configuration out of a process. Since configurations are runtime objects, they do not refer to any free variables and $\mathcal{V}$ is empty. The msg rule is analogous.

6.2.2 *Global Progress.* To state progress, we need to define a *poised process* [43]. A process $\mathsf{proc}(c, P)$ is poised if it is trying to receive a message on c. Dually, a message $\mathsf{msg}(c, M)$ is poised if it is sending along c. Concretely, the following processes are poised:

- $\mathsf{proc}(c, c \leftrightarrow d)$
- $\mathsf{proc}(c, \mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell \in L})$
- $\mathsf{proc}(c, e \leftarrow \mathsf{recv}\ c\ ;\ Q)$
- $\mathsf{proc}(c, \mathsf{wait}\ c\ ;\ Q)$.

Similarly, the following messages are poised:

- $\mathsf{msg}(c, c.k\ ;\ c \leftrightarrow c')$
- $\mathsf{msg}(c, \mathsf{send}\ c\ e\ ;\ c \leftrightarrow c')$
- $\mathsf{msg}(c, \mathsf{close}\ c)$.

A configuration is poised if every message or process in the configuration is poised. Intuitively, this represents that the configuration is trying to communicate *externally* along one of the channels it uses or provides. Note here that for an internal communication to occur on a channel c, either the sending message or the receiving process would be *offering on a channel that is not c*, thus not being poised. This means that for a poised configuration, no internal communication is possible.

Theorem 6.2 (Type Safety). *For a well-typed configuration $\Delta_1 \vDash_C \mathcal{S} :: \Delta_2$,*

(i) (Preservation) If $\mathcal{S} \mapsto \mathcal{S}'$, then $\Delta_1 \vDash_C \mathcal{S}' :: \Delta_2$
(ii) (Progress) Either $\mathcal{S}$ is poised, or $\mathcal{S} \mapsto \mathcal{S}'$.

Proof. Preservation is proved by case analysis on the rules of operational semantics. First, we invert the derivation of the current configuration $\mathcal{S}$ and use the premises to assemble a new derivation for $\mathcal{S}'$. As an illustration, consider the case of typing a spawn: assume that $\mathcal{S} = \mathcal{D}, \mathsf{proc}(c, x \leftarrow f[\theta]\ \overline{d}\ ;\ Q)$. Inverting the comp rule, we obtain $\Delta_1 \Vdash_C \mathcal{D} :: \Delta_1'$ and $\Delta_1' \Vdash_C \mathsf{proc}(c, x \leftarrow f[\theta]\ \overline{d}\ ;\ Q) :: \Delta_2$. Inverting the proc rule on the second premise, we obtain $\Delta, \Delta' \vdash x \leftarrow f[\theta]\ \overline{d}\ ;\ Q :: (c : C)$ where $\Delta, \Delta', \Delta'' = \Delta_1'$ and $\Delta'', (c : C) = \Delta_2$ (implicitly applying the emp rule) for some Δ, Δ', and C. Inverting the def rule on the well-typed process, we obtain $\Delta, (x : \theta(B)) \vdash Q :: (c : C)$ (assuming $\overline{y : B'} \vdash f = P_f :: (x : B)$ is the definition of f).

To obtain the newly formed configuration, we apply the def$C$ semantics rule:

$$\mathrm{proc}(c, x \leftarrow f[\theta]\ \overline{d}\ ;\ Q) \mapsto \mathrm{proc}(a, P_f[a/x][\overline{d}/\overline{y}]), \mathrm{proc}(c, Q[a/x]).$$

From the well typedness of f, we deduce $\Delta' \vdash P_f[a/x][\overline{d}/\overline{y}] :: (a : \theta(B))$, and similarly $\Delta, (a : \theta(B)) \vdash Q[a/x] :: (c : C)$. To reassemble the new configuration, we get $\Delta, \Delta' \Vdash_C \mathrm{proc}(a, P_f[a/x][\overline{d}/\overline{y}]) :: (\Delta, (a : \theta(B)))$ by the proc and emp rules. Composing with Q via the comp rule, we have $\Delta, \Delta' \Vdash_C \mathrm{proc}(a, P_f[a/x][\overline{d}/\overline{y}]), \mathrm{proc}(c, Q[a/x]) :: (c : C)$. Composing with the empty configuration, we conclude $\Delta, \Delta', \Delta'' \Vdash_C \mathrm{proc}(a, P_f[a/x][\overline{d}/\overline{y}]), \mathrm{proc}(c, Q[a/x]) :: (\Delta'', (c : C))$. Remembering that $\Delta, \Delta', \Delta'' = \Delta_1'$ and $\Delta'', (c : C) = \Delta_2$ and composing the preceding with $\mathcal{D}$, we can finally conclude $\Delta_1 \Vdash_C \mathcal{S}' :: \Delta_2$, where $\mathcal{S}' = \mathcal{D}, \mathrm{proc}(a, P_f[a/x][\overline{d}/\overline{y}]), \mathrm{proc}(c, Q[a/x])$.

Progress is proved by induction on the right-to-left typing of $\mathcal{S}$. Note here that the configuration is a multiset that can be split in several ways. Progress relies on a partial ordering among the semantic objects in the configuration. We mandate that the provider of a channel occurs to the left of its client in the configuration. As long as this ordering is maintained, we can choose any split of the configuration. Formally, either $\mathcal{S}$ is empty (and therefore poised) or $\mathcal{S} = (\mathcal{D}, \mathrm{proc}(c, P))$ or $\mathcal{S} = (\mathcal{D}, \mathrm{msg}(c, M))$. By the induction hypothesis, either $\mathcal{D} \mapsto \mathcal{D}'$ or $\mathcal{D}$ is poised. In the former case, $\mathcal{S}$ takes a step (since $\mathcal{D}$ does). In the latter case, we analyze the cases for P and M, applying multiple steps of inversion to show that in each case either $\mathcal{S}$ can take a step or is poised. As an illustration, consider the case where the rightmost semantic object is a process $P = \mathrm{proc}(c, \mathrm{case}\ d\ (\ell \Rightarrow Q_\ell)_{\ell \in L})$. Since $\mathcal{D}$ appears to the left of P, it contains the provider of d, and since $\mathcal{D}$ is well typed, channel d must have an internal choice type. Noting the possibilities for an internal choice, we deduce that the semantic object must be $\mathrm{msg}(c, c.k\ ;\ c \leftrightarrow c')$. Together, we can apply the $\oplus C$ rule so that the whole configuration makes progress. Other cases are analogous. ◻

7 RELATIONSHIP TO CONTEXT-FREE SESSION TYPES

As ordinarily formulated, session types express communication protocols that can be described by regular languages [51]. In particular, the type structure is necessarily tail recursive. CFSTs were introduced by Thiemann and Vasconcelos [51] as a way to express a class of communication protocols that are not limited to tail recursion. CFSTs express protocols that can be described by single-state, real-time DPDAs that use the empty stack acceptance criterion [1, 38].

Despite their name, the essence of CFSTs is not their connection to a particular subset of the (deterministic) context-free languages. Rather, the essence of CFSTs is that session types are enriched to admit a notion of sequential composition. Nested session types are strictly more expressive than CFSTs, in the sense that there exists a proper fragment of nested session types that is closed under a notion of sequential composition. (In keeping with process algebras like ACP [2], we define a sequential composition to be an operation that satisfies the laws of a right-distributive monoid.)

Consider (up to α,β,η-equivalence) the linear, tail functions from types to types with unary type constructors only.

$$\begin{aligned} S, T ::= {} & \hat{\lambda}\alpha.\, \alpha \mid \hat{\lambda}\alpha.\, V[S\,\alpha] \mid \hat{\lambda}\alpha.\, \oplus\{\ell : S_\ell\,\alpha\}_{\ell \in L} \mid \hat{\lambda}\alpha.\, \&\{\ell : S_\ell\,\alpha\}_{\ell \in L} \\ & \mid \hat{\lambda}\alpha.\, A \otimes (S\,\alpha) \mid \hat{\lambda}\alpha.\, A \multimap (S\,\alpha) \end{aligned}$$

The linear, tail nature of these functions allows the type α to be thought of as a continuation type for the session. The functions S are closed under function composition, and the identity function, $\hat{\lambda}\alpha.\, \alpha$, is included in this class of functions. Moreover, because these functions are tail functions, composition right-distributes over the various logical connectives in the following sense.

$$(\hat{\lambda}\alpha.\, V[S\,\alpha]) \circ T = \hat{\lambda}\alpha.\, V[(S \circ T)\,\alpha]$$

$$(\hat{\lambda}\alpha.\, \oplus\{\ell : S_\ell\,\alpha\}_{\ell\in L}) \circ T = \hat{\lambda}\alpha.\, \oplus\{\ell : (S_\ell \circ T)\,\alpha\}_{\ell\in L}$$

$$(\hat{\lambda}\alpha.\, \&\{\ell : S_\ell\,\alpha\}_{\ell\in L}) \circ T = \hat{\lambda}\alpha.\, \&\{\ell : (S_\ell \circ T)\,\alpha\}_{\ell\in L}$$

$$(\hat{\lambda}\alpha.\, A \otimes (S\,\alpha)) \circ T = \hat{\lambda}\alpha.\, A \otimes ((S \circ T)\,\alpha)$$

$$(\hat{\lambda}\alpha.\, A \multimap (S\,\alpha)) \circ T = \hat{\lambda}\alpha.\, A \multimap ((S \circ T)\,\alpha)$$

These distributive properties justify interpreting $S \circ T$ as "T after S" because the $(S \circ T)\,\alpha = S\,(T\,\alpha)$ found on the right-hand sides of these equations "calls" S with the continuation $T\,\alpha$. Together with the monoid laws of function composition, these distributive properties therefore allow us to define sequential composition as $S; T = S \circ T$.

This suggests that although many details distinguish our work from CFSTs, nested session types cover the essence of sequential composition underlying CFSTs. However, even stating a theorem that every CFST process can be translated into a well-typed process in our system of nested session types is difficult because the two type systems differ in many details: we include $\otimes$ and $\multimap$ as session types but CFSTs do not; CFSTs use a complex kinding system to incorporate unrestricted session types and combine session types with ordinary function types; the CFST system uses classical typing for session types and a procedure of type normalization, whereas our types are intuitionistic and do not rely on normalization; and the CFST typing rules are based on natural deduction rather than the sequent calculus. With all of these differences, a formal translation, theorem, and proof would not be very illuminating beyond the essence already described here. Empirically, we can also give analogues of the published examples for CFSTs (e.g., see the first two examples of Section 9).

Finally, nested session types are strictly *more* expressive than CFSTs. Recall from Section 2 that the language $L_3 = \{\mathbf{L}^n\mathbf{a}\,\mathbf{R}^n\mathbf{a} \cup \mathbf{L}^n\mathbf{b}\,\mathbf{R}^n\mathbf{b} \mid n > 0\}$ can be expressed using nested session types with *two* type parameters used in an essential way. Moreover, Korenjak and Hopcroft [38] observe that this language cannot be recognized by a single-state, real-time DPDA that uses empty stack acceptance, and thus CFSTs cannot express the language L_3. More broadly, nested types allow for finitely many states and acceptance by empty stack or final state, whereas the emphasis on sequential composition of types in CFSTs means that they only allow a single state and empty stack acceptance [51].

8 IMPLEMENTATION

We have implemented a prototype for nested session types and integrated it with the open source Rast system [18]. Rast (Resource-aware session types) is a programming language that implements the intuitionistic version of session types [7] with support for arithmetic refinements [19], and ergometric [17] and temporal [16] types for complexity analysis. Our prototype extension is implemented in Standard ML (8,011 lines of code) containing a lexer and parser (1,214 lines), a type checker (3,001 lines), and an interpreter (201 lines), and is well documented. The prototype is available in the Rast repository [13].

8.1 Syntax

A program contains a series of mutually recursive type and process declarations and definitions, concretely written as follows.

```
type V[x1]...[xk] = A
decl f[x1]...[xk] : (c1 : A1) ... (cn : An) |- (c : A)
proc c <- f[x] c1 ... cn = P
```

Type $V[\overline{x}]$ is represented in concrete syntax as `V[x1]...[xk]`. The first line is a *type definition*, where V is the type name parameterized by type variables $x_1, \ldots, x_k$ and A is its definition. The second line is a *process declaration*, where f is the process name (parameterized by type variables $x_1, \ldots, x_k$), $(c_1 : A_1) \ldots (c_n : A_n)$ are the used channels and corresponding types, and the offered channel is c of type A. Finally, the last line is a *process definition* for the same process f defined using the process expression P. We use a handwritten lexer and shift-reduce parser to read an input file and generate the corresponding abstract syntax tree of the program. The reason to use a handwritten parser instead of a parser generator is to anticipate the most common syntax errors that programmers make and respond with the best possible error messages.

Once the program is parsed and its abstract syntax tree is extracted, we perform a *validity check* on it. This includes checking that type definitions, and process declarations and definitions are closed with respect to the type variables in scope. To simplify and improve the efficiency of the type equality algorithm, we also assign internal names to type subexpressions parameterized over their free index variables. These internal names are not visible to the programmer.

8.2 Type Checking and Error Messages

The implementation is carefully designed to produce precise error messages. To that end, we store the extent (source location) information with the abstract syntax tree and use it to highlight the source of the error. We also follow a bidirectional type checking [44] algorithm reconstructing intermediate types starting with the initial types provided in the declaration. This helps us precisely identify the source of the error. Another particularly helpful technique has been *type compression*. Whenever the type checker expands a type $V[\theta]$ defined as $V[\overline{\alpha}] \triangleq B$ to $\theta(B)$, we record a reverse mapping from $\theta(B)$ to $V[\overline{\alpha}]$. When printing types for error messages this mapping is consulted, and complex types may be compressed to much simpler forms, greatly aiding readability of error messages.

9 MORE EXAMPLES

All of our examples have been implemented and type checked in the open source Rast repository [13]. We have also further implemented the standard polymorphic data structures such as lists, stacks, and queues.

9.1 Arithmetic Expression Server

We adapt the example of an arithmetic expression server from prior work on CFSTs [51].

9.1.1 Binary Natural Numbers. Before we can describe the expression server, we need a type `bin` that describes binary natural numbers.

```
type bin = +{ b0 : bin , b1 : bin , $ : 1 }
```

A process that *provides* type `bin` will send a stream of bits, `b0` and `b1`, starting with the least significant bit and eventually ending with `$`.[7]

Given this type, we can define processes `double`, `inc`, and `plus` that double and increment a binary natural number and add two binary numbers, respectively. The `double` process uses a binary natural number and offers another binary natural number that represents double the value.

[7]Strictly speaking, because the interpretation of types is coinductive, `bin` includes potentially infinite binary natural numbers such as the infinite stream of bits `b1`. This will also apply to types in the following examples. Even in the absence of nested types, to make types truly inductive, other machinery would be needed (e.g., [21]).

```
decl double : (n0 : bin) |- (n : bin)
proc n <- double n0 =
  n.b0 ; n <-> n0
```

The `n <-> n0` forwards channel `n0` to channel `n`. Here, forwarding means that messages on one channel are sent along the other channel. Because it functions as a computational step and is not merely a static renaming, we cannot avoid including it in the process code. (This holds for all other forwarding steps in the following processes.)

The `inc` process uses a binary natural number and offers another binary natural number that represents the incremented value.

```
decl inc : (n0 : bin) |- (n : bin)
proc n <- inc n0 =
  case n0 (
    $ => n.b1 ; n.$ ; n <-> n0
  | b0 => n.b1 ; n <-> n0
  | b1 => n.b0 ; n <- inc n0 )
```

The `plus` process uses two binary natural numbers and offers another binary natural number that represents their sum. In one case, it calls the `inc` process.

```
decl plus : (n1 : bin) (n2 : bin) |- (n : bin)
proc n <- plus n1 n2 =
  case n1 (
    $ => wait n1 ; n <-> n2
  | b0 => case n2 (
            $ => wait n2 ; n.b0 ; n <-> n1
          | b0 => n.b0 ; n <- plus n1 n2
          | b1 => n.b1 ; n <- plus n1 n2 )
  | b1 => case n2 (
            $ => wait n2 ; n.b1 ; n <-> n1
          | b0 => n.b1 ; n <- plus n1 n2
          | b1 => n.b0 ; n' <- plus n1 n2 ; n <- inc n' ) )
```

9.1.2 Arithmetic Expressions in Prefix Notation. From binary natural numbers, we can construct a simple language of arithmetic expressions supporting doubling and addition operations. When written in prefix notation, these expressions are described by the type `exp[K]`. More precisely, the type `exp[K]` describes an expression followed by a suffix, or *continuation*, of type `K`.

```
type exp[K] = +{ const : bin * K , dbl : exp[K] , add : exp[exp[K]] }
```

An expression is either a constant, a doubling operation applied to an expression, or an addition operation applied to two expressions. But in all cases, a suffix, or continuation, of type `K` follows the expression, as enforced by type nesting:

- If a process providing type `exp[K]` sends the `const` label, then it sends a binary number of type `bin` and continues as type `K`.
- If that process sends the `dbl` label, then it continues as type `exp[K]`, ultimately delivering an expression followed by a suffix of type `K`.
- If that process sends the `add` label, then it continues as type `exp[exp[K]]`. In other words, it continues by delivering an expression followed by a suffix of type `exp[K]`, which is itself

an expression followed by a suffix of type K, and these two expressions are exactly the two summands.

As an illustration, consider two binary constants a and b, and suppose that we want to create the expression $a + 2b$. Written in prefix notation, this expression is $+\ a\ (\times 2\ b)$, where $\times 2$ denotes the doubling operation.We can build this expression in its prefix notation (followed by a suffix of type K) as the following `example[K]` process, parameterized by type K. (In the following code, we provide the intermediate typing judgments in comments along the right.)

```
decl example[K] : (a : bin) (b : bin) (k : K) |- (e : exp[K])
proc e <- example[K] a b k =
  e.add ;                % (a:bin) (b:bin) (k:K) |- (e : exp[exp[K]])
  e.const ; send e a ;   % (b:bin) (k:K) |- (e : exp[K])
  e.dbl ;                % (b:bin) (k:K) |- (e : exp[K])
  e.const ; send e b ;   % (k:K) |- (e : K)
  e <-> k
```

Imitating the prefix notation $+\ a\ (\times 2\ b)$, this process sends the add label, followed by the `const` label and the binary natural number a, followed by labels `dbl` and `const` and the binary natural number b. Finally, the process continues at type K by forwarding k to e.

9.1.3 Evaluation Server. To evaluate a term, we can define an `eval` process, parameterized by the type K.

```
decl eval[K] : (e : exp[K]) |- (v : bin * K)
```

The `eval` process uses a channel e of type `exp[K]` and offers a channel v of type `bin * K`. The process evaluates expression e and sends its binary value together with the continuation of type K along channel v. The process `eval[K]` is defined by the following.

```
proc v <- eval[K] e =
  case e (
    const => x <- recv e ;                % (x:bin) (e:K) |- (v : bin * K)
      send v x ; v <-> e
  | dbl =>                                % (e : exp[K]) |- (v : bin * K)
      v' <- eval[K] e ; x <- recv v' ;    % (x:bin) (v':K) |- (v : bin * K)
      y <- double x ;                     % (y:bin) (v':K) |- (v : bin * K)
      send v y ; v <-> v'
  | add =>                                % (e:exp[exp[K]]) |- (v : bin*K)
      v1 <- eval[exp[K]] e ; x <- recv v1 ; % (x:bin) (v1:exp[K]) |- (v:bin*K)
      v2 <- eval[K] v1 ; y <- recv v2 ;   % (y:bin) (v2:K) |- (v : bin * K)
      z <- plus x y ;                     % (z:bin) (v2:K) |- (v : bin * K)
      send v z ; v <-> v2 )
```

Evaluation begins by analyzing the shape of expression e (in each branch, a forwarding process is necessary to connect the remaining channel being used to the channel being provided):

- If the expression begins with `const`, then the binary constant that follows is sent along channel v as the expression's value.
- If the expression instead begins with `dbl`, then the subsequent expression is itself evaluated by a recursive call to `eval[K]`. The resulting value is doubled and then sent along channel v.
- Otherwise, if the expression begins with add, then the subsequent expression is evaluated by a recursive call to `eval[exp[K]]`. Notice that this recursive call is nested, being made

at suffix type exp[K]. This gives the first summand's value, together with a suffix of type exp[K], which is the expression corresponding to the second summand. The second summand is then evaluated by another recursive call, this time eval[K] at suffix type K. The two values are added together by a call to plus and finally sent along channel v as the expression's overall value.

As can be seen in the first recursive call to eval, at type exp[K], the additional generality provided by nested recursion is crucial here.

9.2 Serializing Binary Trees

Another example from Thiemann and Vasconcelos [51] is that of serializing binary trees. Here we adapt that example to our system.

9.2.1 Binary Trees. Binary trees can be described by the following type.

```
type Tree[a] = +{ node : Tree[a] * (a * Tree[a]) , leaf : 1 }
```

These trees are polymorphic in the type a of data stored at each internal node. A tree is either an internal node or a leaf, with the internal nodes storing channels that emit the left subtree, data, and right subtree.

In what follows, it will sometimes be useful to have *processes* for constructing trees and pairs. These node, leaf, and pair processes are defined as follows.

```
decl node[a] : (l : Tree[a]) (x:a) (r : Tree[a]) |- (t : Tree[a])
proc t <- node[a] l x r =
  t.node ; send t l ; send t x ; t <-> r

decl leaf[a] : . |- (t : Tree[a])
proc t <- leaf[a] =
  t.leaf ; close t

decl pair[a][b] : (x:a) (y:b) |- (p : a * b)
proc p <- pair[a][b] x y =
  send p x ; p <-> y
```

Notice that, owing to the several channels stored at each node for the left subtree, data, and right subtree, these Tree[a] trees do not exist *a priori* in a serial form.

9.2.2 Serialized Binary Trees. We can, however, use a different type to represent serialized trees. The type constructor STree[a][K] is defined over two parameters: the parameter a for the type of data, and the parameter K for the type of the suffix, or continuation, that will follow the serialized tree.

```
type STree[a][K] = +{ nd : STree[a][a * STree[a][K]] ,
                      lf : K }
```

A serialized tree is then a stream of node and leaf labels, nd and lf, parameterized by a suffix type K. Like add in the expression server, the label nd continues with type STree[a][a * STree[a][K]]: the label nd is followed by the serialized left subtree, which itself continues by sending the data stored at the internal node and then the serialized right subtree, which continues with type K.[8]

[8]The presence of a * means that, in the strictest sense, this is not a true serialization because it sends a separate channel along which the data of type a is emitted. But there is no uniform mechanism for serializing polymorphic data, so this is

9.2.3 Serializing and Deserializing Binary Trees. Using these types, it is relatively straightforward to implement processes that serialize and deserialize such trees. The process `serialize` has the following type.

```
decl serialize[a][K] : (t : Tree[a]) (k : K) |- (s : STree[a][K])
```

This process uses channels `t` and `k` that hold the tree and the continuation, respectively, and offers the corresponding serialized tree along channel `s`. Notice that this is parametric in the type `K` of the continuation; this polymorphism will be essential.

The process `serialize` can be defined as follows.

```
proc s <- serialize[a][K] t k =
  case t (
    node =>                       % (t : Tree[a] * (a * Tree[a])) ... |- ...
      l <- recv t ; x <- recv t ; % (l:Tree[a]) (x:a) (t:Tree[a]) ... |- ...
      s.nd ;                      % ... |- (s : STree[a][a * STree[a][K]])
      sr <- serialize[a][K] t k ; % ... (sr : STree[a][K]) |- ...
      p <- pair[a][STree[a][K]] x sr ;    % ... (p : a * STree[a][K]) |- ...
      s <- serialize[a][a * STree[a][K]] l p
  | leaf =>                       % (t : 1) (k : K) |- (s : STree[a][K])
      s.lf ;                      % ... |- (s : K)
      wait t ; s <-> k )
```

In the preceding code comments, we use `...` to elide the types of those channels that remain unchanged by the preceding line of code.

Serialization begins by examining the tree's root, which is either a `node` or a `leaf`:

- If the root is a `node`, then the corresponding serialized tree begins with `nd`, and type `STree[a][a * STree[a][K]]` must be offered along channel `s`. To do so, we begin with a recursive call to `serialize` that serves to serialize the right subtree with the given continuation, `k:K`, forming an `STree[a][K]`. This serialized right subtree is then paired with the data `x:a` via a call to `pair[a][STree[a][K]]`. A subsequent recursive call serializes the left subtree, using the pair `p` of the data and the serialized right subtree as the new continuation; this forms an `STree[a][a * STree[a][K]]`, just as required.
- If the tree is only a leaf, then the process forwards to the continuation.

The process `deserialize` for deserializing binary trees has the following type.

```
decl deserialize[a][K] : (s : STree[a][K]) |- (tk : Tree[a] * K)
```

This process uses a channel `s` that holds a serialized binary tree (and its continuation of type `K`) and offers the corresponding deserialized tree along channel `tk`. Once again, this type is parametric in the continuation type, `K`, which is essential to implementing the process in a well-typed way. The process `deserialize[a][K]` can be defined as follows.

```
proc tk <- deserialize[a][K] s =
  case s (
    nd =>                          % (s : STree[a][a * STree[a][K]]) |- ...
      s' <- deserialize[a][a * STree[a][K]] s ;  % (s' : Tree[a] * (a * STree[a][K])) |- ...
      l <- recv s' ;               % (l:Tree[a]) (s': a * STree[a][K]) |- ...
      x <- recv s' ;               % ... (x:a) (s':STree[a][K]) |- ...
      s" <- deserialize[a][K] s' ; % ... (s" : Tree[a] * K) |- ...
      r <- recv s" ;               % ... (r:Tree[a]) (s":K) |- ...
      t <- node[a] l x r ;         % ... (t:Tree[a]) |- ...
      send tk t ; tk <-> s"
```

as close to a true serialization as possible. Concrete instances of type `Tree` with, say, data of base type `int` could be given a true serialization by "inlining" the data of type `int` in the serialization.

```
    | lf =>                         % (s:K) |- ...
        t <- leaf[a] ;              % ... (t:Tree[a]) |- ...
        send tk t ; tk <-> s )
```

To deserialize a serialized tree, the first step is to analyze the beginning of the serialized form. It must begin with either `nd` or `lf`, the serialized forms of nodes and leaves:

- If it begins with `nd`, then what follows is an `STree[a][a * STree[a][K]]`—that is, a serialized left subtree, followed by a pair of the internal node's data together with a serialized right subtree. A recursive call at continuation type `a * STree[a][K]` allows us to reconstruct the left subtree, and from the continuation, we extract the node's data. Then what remains is the serialized right subtree of type `STree[a][K]`. Another recursive call, this time at continuation type `K`, allows us to reconstruct the right subtree. A call to `node[a]` rebuilds the entire tree, which is then sent along channel `tk`.
- If the serialized form instead begins with `lf`, then it represents a leaf and what follows is just the continuation of type `K`. A (deserialized) leaf is constructed by a call to `leaf[a]`, and it is then sent along channel `tk` as the deserialized tree.

9.3 Generalized Tries for Binary Trees

Using nested types in Haskell, prior work [31] describes an implementation of generalized tries that represent mappings on binary trees. Our nested session type system is also expressive enough to represent such generalized tries. Without the nested session types that our work introduces to Rast, it would not be possible to cleanly represent generalized tries in Rast.

9.3.1 Tries. We can reuse the type `Tree[a]` of binary trees given earlier. The type `Trie[a][b]` describes tries that represent mappings from keys of type `Tree[a]` to values of type `b`.

```
type Trie[a][b] = &{ lookup_leaf : b ,
                     lookup_node : Trie[a][a -o Trie[a][b]] }
```

Unlike the types in the previous examples, the type `Trie[a][b]` is an external, not internal, choice. A process that provides type `Trie[a][b]` offers its client a choice of two operations: `lookup_leaf`, which returns a value of type `b` that the mapping assigns to a leaf, and `lookup_node`, which returns a trie in which the node's left subtree (and subsequently, right subtree) can be looked up. This type will become more clear as we describe how to look up a tree in a trie.

9.3.2 Looking Up a Tree in a Trie. A process for looking up a tree in such tries can be declared by the following.

```
decl lookup_tree[a][b] : (m : Trie[a][b]) (t : Tree[a]) |- (v : b)
```

This process uses channels `m` and `t` that hold the trie and the tree to look up, respectively, and offers the corresponding value of type `b` along channel `v`. The process `lookup_tree` can be defined as follows.

```
proc v <- lookup_tree[a][b] m t =
  case t (
    leaf => m.lookup_leaf ;                         % (m : b) (t : 1) |- ...
      wait t ; v <-> m
  | node =>                                         % ... (t : Tree[a] * (a * Tree[a])) |- ...
      l <- recv t ; x <- recv t ;                   % ... (l:Tree[a]) (x:a) (t:Tree[a]) |- ...
      m.lookup_node ;                               % (m : Trie[a][a -o Trie[a][b]]) ... |- ...
      m' <- lookup_tree[a][a -o Trie[a][b]] m l ;   % (m' : a -o Trie[a][b]) ... |- ...
      send m' x ;                                   % (m' : Trie[a][b]) ... |- ...
      v <- lookup_tree[a][b] m' t )
```

To look up a tree in a trie, first determine whether that tree is a `leaf` or a `node`:

- If the tree is a `leaf`, then sending `lookup_leaf` to the trie will return the value of type `b` associated with that tree in the trie.
- Otherwise, if the tree is a `node`, then sending `lookup_node` to the trie results in `Trie[a][a -o Trie[a][b]]` that represents a mapping from left subtrees to values of type `a -o Trie[a][b]`. We can look up the left subtree in this trie, resulting in a process that offers type `a -o Trie[a][b]`. To this process, we then send the data stored at the original tree's root. That results in a trie of type `Trie[a][b]` that represents a mapping from right subtrees to values of type `b`. Therefore, we finally look up the right subtree in this new trie and obtain a value of type `b`, as desired.

9.3.3 Building a Trie from a Total Function on Trees. In the tree serialization example, we were able to define `deserialize` as an inverse to `serialize`. Similarly, as an inverse to `lookup_tree`, we can define a process `build_trie` that constructs a trie from a (total, linear) function on trees.

```
decl build_trie[a][b] : (f : Tree[a] -o b) |- (m : Trie[a][b])
```

Both `lookup_tree` and `build_trie` can be seen as analogues to `deserialize` and `serialize`, respectively, converting a lower-level representation to a higher-level representation and vice versa. These types and declarations mean that tries represent total mappings; partial mappings are also possible, at the expense of some additional complexity in the type and process definitions.

The `build_trie` process can be defined as follows.

```
proc m <- build_trie[a][b] f =
  case m (
    lookup_leaf =>            % ... |- (m : b)
      t <- leaf[a] ;          % ... |- (t : Tree[a])
      send f t ; m <-> f
  | lookup_node =>            % ... |- (m : Trie[a][a -o Trie[a][b]])
      g <- fn_left[a][b] f ;  % (g : Tree[a] -o (a -o Trie[a][b])) |- ...
      m <- build_trie[a][a -o Trie[a][b]] g )
```

The trie constructed by `build_trie` waits to receive either a `lookup_leaf` or `lookup_node` label as an instruction:

- If `lookup_leaf` is received, then this trie process constructs a `leaf`. The value that function `f` assigns to a `leaf` is looked up, then forwarded to the trie's client.
- Otherwise, if `lookup_node` is received, a trie of type `Trie[a][a -o Trie[a][b]]` must be constructed. That can be done by making a recursive call to `build_trie` at the type `a -o Trie[a][b]`, so long as there is a function of type `Tree[a] -o (a -o Trie[a][b])` that maps left subtrees to functions of type `a -o Trie[a][b]`. That function is provided by a named helper process, `fn_left`.

The helper process `fn_left` is defined as follows.

```
decl fn_left[a][b] : (f : Tree[a] -o b) |- (g : Tree[a] -o (a -o Trie[a][b]))
proc g <- fn_left[a][b] f =
  l <- recv g ; x <- recv g ; % ... (l:Tree[a]) (x:a) |- (g : Trie[a][b])
  h <- fn_right[a][b] l x f ; % (h : Tree[a] -o b) |- ...
  g <- build_trie[a][b] h
```

This process uses a function `f` that maps trees to values of type `b`, and offers a function that maps left subtrees and a datum of type `a` to tries that map right subtrees to values of type `b`. The helper process `fn_left` first inputs a left subtree and a datum of type `a`, then makes a (morally) recursive call to `build_trie` to construct a trie. This call requires a function of type `Tree[a] -o b` that maps right subtrees to values of type `b`—this is the purpose of the `fn_right` helper process.

The helper process `fn_right` is defined as follows.

```
decl fn_right[a][b] : (l:Tree[a]) (x:a) (f:Tree[a] -o b) |- (h:Tree[a] -o b)
proc h <- fn_right[a][b] l x f =
  r <- recv h ;        % ... (r : Tree[a]) |- (h : b)
  t <- node[a] l x r ; % ... (t : Tree[a]) |- ...
  send f t ; h <-> f
```

This process takes a left subtree, a datum of type `a`, and a function that maps trees to values of type `b`, using these to construct a mapping from right subtrees to values of type `b`. The process receives a right subtree and, by calling the `node` process, puts it together with a left subtree and datum to form a tree. This tree is passed to the function `f` to obtain the corresponding value of type `b`.

The current Rast implementation does not support anonymous process calls (although they could be added in a straightforward way). For this reason, we depend on the named helper processes `fn_left` and `fn_right`. In a language with anonymous processes, the bodies of `fn_left` and `fn_right` could easily be inlined.

9.4 Queues

Here we elaborate on the queue example from Section 2.

9.4.1 Basic Type. At a basic level, polymorphic queues holding data of type `a` can be described by the type:

```
type Queue'[a] = &{ enq: a -o Queue'[a] , deq: Option[a][Queue'[a]] },
```

where

```
type Option[a][k] = +{ some: a * k , none: 1 }.
```

Each queue supports enqueue and dequeue operations with an external choice between `enq` and `deq` labels. If the queue's client chooses `enq`, then the subsequent type, `a -o Queue'[a]`, requires that the client send an `a`; then the structure recurs at type `Queue'[a]` to continue serving enqueue and dequeue requests. If the queue's client instead chooses `deq`, then the subsequent type, `Option[a][Queue'[a]]`, requires the client to branch on whether the queue is non-empty—whether there is `some` datum or `none` at all at the front of the queue.

9.4.2 Type Nesting Enforces an Invariant. Implicit in this description of how a queue would offer type `Queue'[a]` is a key invariant about the queue's size: dequeuing from a queue into which an element was just enqueued should always yield `some` element, never `none` at all. However, the type `Queue'[a]` cannot enforce this dequeue-after-enqueue invariant precisely because it does not track the queue's size—`Queue'[a]` can be used equally well to type empty queues as to type queues containing three elements, for instance. But by taking advantage of the expressive power provided by nested types, we can enforce the invariant by defining a type `Queue[a][k]` of `k`-sized queues that can enforce the dequeue-after-enqueue invariant.

We will start by defining two types that describe, in a somewhat elaborated way, sizes.

```
type Some[a][k] = +{ some: a * k }
type None = +{ none: 1 }
```

The type `Some[a][k]` describes a `k` with `some` element of type `a` added at the front; the type `None` describes an empty shape. These types function similarly to unary natural numbers: `Some` acts like a successor for natural numbers, and None acts like zero for natural numbers. In this way, these types express the size of a queue.

The idea is that Queue[a][None] will type empty stacks, because they have the shape None, whereas the type Queue[a][Some[a][Queue[a][None]]] will represent queues containing one element, because they have Some element in front of an empty queue, and so on for queues of larger sizes.

More generally, the type Queue[a][k] describes k-sized queues.

```
type Queue[a][k] = &{ enq: a -o Queue[a][Some[a][Queue[a][k]]] , deq: k }
```

Once again, each queue supports enqueue and dequeue operations with an external choice between enq and deq labels. This time, however, enqueuing an a into the queue leads to type Queue[a][Some[a][Queue[a][k]]] (i.e., a queue with Some element in front of a k-sized queue). Equally importantly, dequeuing from a k-sized queue exposes the "size" k.

Together, these two aspects of the type Queue[a][k] serve to enforce the dequeue-after-enqueue invariant. Suppose that a client enqueues an a into a queue q of type Queue[a][k]. After the enqueue, the queue q will have type Queue[a][Some[a][Queue[a][k]]]. If the client then dequeues from q, the type becomes Some[a][Queue[a][k]], which is +{ some: a * Queue[a][k] }. This means that the client will always receive some element, never none at all because none is not part of this type. And that is how the type Queue[a][k] enforces the dequeue-after-enqueue invariant.

Given this type constructor, the empty queue can be expressed as a process that has type Queue[a][None], and we can define a process elem[a][k] that constructs a queue of shape Some[a][Queue[a][k]] from an a and a queue of size k.

```
decl empty[a] : . |- (q : Queue[a][None])
decl elem[a][k] : (x:a) (r:Queue[a][k]) |- (q:Queue[a][Some[a][Queue[a][k]]])
```

The empty[a] process is defined as follows.

```
proc q <- empty[a] =
  case q (
    enq => x <- recv q ;     % (x:a) |- (q:Queue[a][Some[a][Queue[a][None]]])
      e <- empty[a] ;        % ... (e : Queue[a][None]) |- ...
      q <- elem[a][None] x e
  | deq =>                   % . |- (q : None)
      q.none ; close q )
```

The empty queue waits to receive either the enq or deq label:

- If the empty queue receives enq, then it inputs a channel x along which an element of type a is offered. A new empty queue is created along a fresh channel e by recursively calling empty[a]. By calling the elem[a][None] (notice the use of None) with channels x and e, the element is placed at the front of the queue.
- Otherwise, if the empty queue receives deq, then it indicates that the queue is empty by sending label none and closing the channel.

The elem[a][k] process is defined as follows.

```
proc q <- elem[a][k] x r =
  case q (
    enq => y <- recv q ;  % ... (y:a) |- (q:Queue[a][Some[a][
```

```
                          %                    Queue[a][Some[a][
                          %                      Queue[a][k]]]]])
      r.enq ; send r y ;  % (r : Queue[a][Some[a][Queue[a][k]]]) |- ...
      q <- elem[a][Some[a][Queue[a][k]]] x r
   | deq =>               % ... |- (q : Some[a][Queue[a][k]])
      q.some ; send q x ; % ... |- (q : Queue[a][k])
      q <-> r )
```

This process also waits to receive either an `enq` or a `deq` label:

- If the process receives `enq`, then it first inputs a channel `y` along which the element to be enqueued is offered. This element is enqueued into the tail of the queue, along channel `r`, by sending label `enq` and channel `y`. This causes the type of channel `r` to become `Queue[a][Some[a][Queue[a][k]]]`. A recursive call to `elem`, this time at the larger size `Some[a][Queue[a][k]]`, is made to ensure that the front of the queue remains unchanged.
- Otherwise, if the process receives label `deq`, then it sends the client the label `some` and the element `x`.

Because this invariant is quite strong, and yet not as easily manipulated as, say, a type that uses arithmetic refinements, it can be difficult to implement certain operations on queues using this type. For this reason, in ongoing work [15], we are developing a notion of subtyping and a sound (but incomplete) subtype checking algorithm. There subtyping allows us to use the more precise `Queue[a][Queue'[a]]` type when possible and revert to the more general supertype `Queue'[a]` as needed.

It is also worth pointing out that the type `Queue[a][k]` applies equally well to stacks. At first, that seems somewhat surprising, given that queues and stacks differ in where they place incoming data and that we use the type parameter `k` to track the queue. But because `k` is essentially an elaboration of the queue's size, and because the sizes of stacks and queues grow in the same way, it perhaps should not be surprising after all.

9.5 Dyck Language of Well-Balanced Parentheses

Recall from Section 2 the example of the Dyck language of well-balanced parentheses. Here we expand upon that example, writing processes that (i) wrap an additional pair of parentheses around a given Dyck word to form another Dyck word and (ii) concatenate two given Dyck words to form another Dyck word.

9.5.1 Types. Recall from Section 2 that the following types are used to describe strings of well-balanced parentheses. (Once again, we use `L` and `R` to stand in for left and right parentheses.)

```
type D0 = +{ L : D[D0] , $ : 1 }
type D[k] = +{ L : D[D[k]] , R : k }
```

The type `D0` describes strings of well-balanced parentheses (followed by a terminal `$`). The type `D[k]` describes "strings of slightly left-unbalanced parentheses," if you will, followed by a continuation of type `k`—that is, strings of well-balanced parentheses that are followed by exactly one more closing parenthesis (i.e., an `R`) and a continuation of type `k`.

9.5.2 Overview of `wrap` *and* `append`*.* The process, `wrap`, that forms a Dyck word by wrapping an additional pair of parentheses around a given Dyck word therefore has the following type.

```
decl wrap : (w : D0) |- (w' : D0)
```

Similarly, the process, `append`, that concatenates two given Dyck words has the following type.

```
decl append : (w1 : D0) (w2 : D0) |- (w' : D0)
```

Conceptually, these wrap and append operations share something in common: both operations rely on placing something at the end of a Dyck word. In the case of wrap, we need to place a right parenthesis at the end of the given word (after having first placed a left parenthesis at the beginning of the word); in the case of append, we need to place a string of balanced parentheses at the end of the given word.

Before delving into the specifics of wrap and append, it will be useful to think about how we might place something at the end of a string of type D[k] (i.e., a string of slightly left-unbalanced parentheses).

9.5.3 Functor Map. Notice that in the type D[k], the continuation of type k always marks the end of the slightly left-unbalanced string. Therefore, we could add something to the end of the slightly left-unbalanced string by changing the continuation used. In other words, what we need is a process of type

```
decl fmap[k][k'] : (f : k -o k') (w : D[k]) |- (w' : D[k'])
```

that copies the sequence of parentheses from the word w to the word w', but when the continuation of type k is reached, the function f is applied to convert the continuation to one of type k'.

This functor map process, fmap[k][k'], is defined as follows.

```
proc w' <- fmap[k][k'] f w =
  case w (
    L => w'.L ;                   % ... (w : D[D[k]]) |- (w' : D[D[k']])
      g <- lift[k][k'] f ;        % (g : D[k] -o D[k']) ... |- ...
      w' <- fmap[D[k]][D[k']] g w
  | R => w'.R ;                   % ... (w : k) |- (w' : k')
      send f w ; w' <-> f )
```

The fmap[k][k'] process begins by examining the first symbol in the word w:

- If the first symbol is a left parenthesis, then it is copied to the word w'. Then we need to construct a process of type (f : k -o k') (w : D[D[k]]) |- (w' : D[D[k']]) that copies from word w to word w' all of the parentheses represented by the nested constructors D[D[-]] and applies the function f to convert the continuation of type k into one of type k'. Suppose that we have a helper process, lift[k][k'], to lift the function f to a function g of type D[k] -o D[k'] that copies the parentheses represented by D[-] and applies f to the continuation of type k. Then a recursive call to fmap[D[k]][D[k']] with function g completes the goal.
- Otherwise, if the first symbol is a right parenthesis, then it is copied to the word w'. The function f is applied to the continuation of type k that follows this right parenthesis, transforming the continuation into one of type k'.

We still need to define the helper process lift[k][k'], however. It is just the abstracted form of a recursive call to fmap[k][k'].

```
decl lift[k][k'] : (f : k -o k') |- (g : D[k] -o D[k'])
proc g <- lift[k][k'] f =
  w <- recv g ; g <- fmap[k][k'] f w
```

Once again, if the Rast implementation supported anonymous processes, then we could alternatively inline this abstraction into the body of fmap itself.

9.5.4 The `wrap` *Process.* With `fmap` in hand, we can finally define the `wrap` process. The new word `w'` begins with a left parenthesis (i.e., `L`), with the remainder of `w'` being exactly the given word `w` with a right parenthesis tacked onto the end:

```
decl wrap : (w : D0) |- (w' : D0)
proc w' <- wrap w =
  w'.L ; w' <- snocR w,
```

where the `snocR` process is defined recursively together with the following `snocR'` helper process.

```
decl snocR : (w : D0) |- (w' : D[D0])
proc w' <- snocR w =
  case w (
    L => w'.L ;                    % (w : D[D0]) |- (w' : D[D[D0]])
      f <- snocR' ;                % ... (f : D0 -o D[D0]) |- ...
      w' <- fmap[D0][D[D0]] f w
  | $ => w'.R ; w'.$ ;             % (w : 1) |- (w' : 1)
      wait w ; close w' )

decl snocR' : . |- (f : D0 -o D[D0])
proc f <- snocR' =
  w <- recv f ; f <- snocR w
```

The `snocR` and its `snocR'` helper very much follow the pattern laid down by `fmap` and `lift`:

- If the first symbol of `w` is a left parenthesis, it is copied to the word `w'`. We then need to construct a process of type `(w : D[D0]) |- (w' : D[D[D0]])` that copies `w` and tacks a right parenthesis onto the end. Using a call to `fmap[D0][D[D0]]`, we can copy the parentheses represented by the outer `D[-]` and rely on the function that is given to `fmap` to tack on a final right parenthesis. The helper `snocR'`, the abstracted form of `snocR`, is just such a function.
- Otherwise, if the first symbol is the terminal `$`, then we can directly insert a final right parenthesis.

9.5.5 The `append` *Process.* The process of concatenating two Dyck words to form a new Dyck word is quite similar to `wrap`. The code is nearly the same, except that a function for appending a Dyck word is used in place of `snocR'`.

```
decl append : (w1 : D0) (w2 : D0) |- (w' : D0)
proc w' <- append w1 w2 =
  case w1 (
    L => w'.L ;                 % (w1 : D[D0]) ... |- (w' : D[D0])
      f <- append' w2 ;         % ... (f : D0 -o D0) |- ...
      w' <- fmap[D0][D0] f w1
  | $ =>                        % (w1 : 1) ... |- ...
      wait w1 ; w' <-> w2 )

decl append' : (w2 : D0) |- (f : D0 -o D0)
proc f <- append' w2 =
  w1 <- recv f ; f <- append w1 w2
```

10 FURTHER RELATED WORK

After a review of the literature, to the best of our knowledge, our work is the first proposal of polymorphic recursion using nested type definitions in session types. Thiemann and Vasconcelos [51] use polymorphic recursion to update the channel between successive recursive calls but do not allow type constructors or nested types. An algorithm to check type equivalence for the non-polymorphic fragment of CFSTs has been proposed by Almeida et al. [1].

Other forms of polymorphic session types have also been considered in the literature. Gay [26] studies bounded polymorphism associated with branch and choice types in the presence of subtyping. He mentions recursive types (which are used in some examples) as future work but does not mention parametric type definitions or nested types. Bono and Padovani [4, 5] propose (bounded) polymorphism to type the endpoints in copyless message-passing programs inspired by session types, but they do not have nested types. Following the approach of Kobayashi [36], Dardha et al. [12] provide an encoding of session types relying on linear and variant types and present an extension to enable parametric and bounded polymorphism (to which recursive types were added separately [11]) but not parametric type definitions nor nested types. Caires et al. [6] and Pérez et al. [42] provide behavioral polymorphism and a relational parametricity principle for session types but without recursive types or type constructors.

Nested session types bear important similarities with first-order cyclic terms, as observed by Jančar. Jančar [33] proves that the trace equivalence problem of first-order grammars is decidable, following the original ideas by Stirling [49] for the language equality problem in DPDAs. These ideas were also reformulated by Sénizergues [47]. Henry and Sénizergues [30] proposed the only practical algorithm to decide the language equivalence problem on DPDAs that we are aware of. Preliminary experiments show that such a generic implementation, even if complete in theory, is a poor match for the demands of our type checker.

On the technical front, our type equality algorithm builds on prior works on coinduction. Coinductive Logic Programming (CoLP) [29] lays the foundation for the loop detection mechanism (def rule in Figure 2) of our algorithm. CoLP deduces the current goal by computing the most general unifier that matches the current call with a call made earlier. However, CoLP still suffers from backtracking that is cleverly avoided by our algorithm through an internal renaming pass before the type equality check. CoLP also does not provide an algorithm to compute this most general unifier. Several heuristics have also been proposed to generalize the coinductive hypothesis in the form of Horn clauses [24] and guarded higher-order fixpoint terms [37]. In contrast to the aforementioned algorithmic variants (including ours) of coinductive proofs, Roşu and Lucanu [45] provide a proof-theoretical foundation of circular coinduction. They devise a three-rule system to derive circular coinductive proofs and prove that this proof system is behaviorally sound.

11 CONCLUSION

Nested session types extend binary session types with parameterized type definitions. This extension enables us to express polymorphic data structures just as naturally as in functional languages. The proposed types are able to capture sequences of communication actions described by deterministic context-free languages recognized by DPDAs with several states, which accept by empty stack or by final state. In this setting, we show that type equality is decidable. To offset the complexity of type equality, we give a practical type equality algorithm that is sound and efficient but incomplete.

In ongoing work, we have been exploring subtyping for nested types. Since the language inclusion problem for simple languages is undecidable [23], the subtyping problem for nested types is also undecidable [15]. However, despite this negative result, we have been working on an algorithm to approximate subtyping. A subtyping relation increases significantly the programs that can be type checked in the system.

In another direction, since Rast [18] supports arithmetic refinements for lightweight verification, it would be interesting to explore how refinements interact with polymorphic type parameters, namely in the presence of subtyping. We would also like to explore examples where the current type equality is not adequate. Perhaps relatedly, we would like to find out if nested session types can express interesting non-trivial properties of distributed protocols such as consensus or leader election (Raft, Paxos, etc.) that might need unbounded memory.

Finally, Keizer et al. [35] describe a coalgebraic view of session types. It would be interesting to examine whether our use of bisimulation could be reframed using their coalgebraic view and, for example, whether the translation found in Section 4.2 could be seen as a full functor from session coalgebras to coalgebras that correspond to grammars.

ACKNOWLEDGMENTS

We wish to express our gratitude to the anonymous reviewers of this and an earlier version of this article for their perceptive comments.

REFERENCES

[1] Bernardo Almeida, Andreia Mordido, and Vasco T. Vasconcelos. 2020. Deciding the bisimilarity of context-free session types. In Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, Vol. 12079. Springer, 39–56. https://doi.org/10.1007/978-3-030-45237-7_3

[2] J. A. Bergstra and J. W. Klop. 1989. ACPτ : A universal axiom system for process specification. In *Algebraic Methods: Theory, Tools and Applications*, Martin Wirsing and Jan A. Bergstra (Eds.). Springer, Berlin, Germany, 445–463. https://doi.org/10.1007/BFb0015048

[3] Richard S. Bird and Lambert G. L. T. Meertens. 1998. Nested datatypes. In *Mathematics of Program Construction*. Lecture Notes in Computer Science, Vol. 1422. Springer, 52–67. https://doi.org/10.1007/BFb0054285

[4] Viviana Bono and Luca Padovani. 2011. Polymorphic endpoint types for copyless message passing. In *Proceedings of the 4th Interaction and Concurrency Experience (ICE'11)*. 1–19. https://doi.org/10.4204/EPTCS.59.5

[5] Viviana Bono and Luca Padovani. 2012. Typing copyless message passing. *Logical Methods in Computer Science* 8, 1 (2012), 17. https://doi.org/10.2168/LMCS-8(1:17)2012

[6] Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. 2013. Behavioral polymorphism and parametricity in session-based communication. In *Programming Languages and Systems*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, Berlin, Germany, 330–349. https://doi.org/10.1007/978-3-642-37036-6_19

[7] Luís Caires and Frank Pfenning. 2010. Session types as intuitionistic linear propositions. In *Concurrency Theory*. Lecture Notes in Computer Science, Vol. 6269. Springer, 222–236. https://doi.org/10.1007/978-3-642-15375-4_16

[8] Luís Caires, Frank Pfenning, and Bernardo Toninho. 2014. Linear logic propositions as session types. *Mathematical Structures in Computer Science* 760, 5 (2014), 1–55. https://doi.org/10.1017/S0960129514000218

[9] Iliano Cervesato and Andre Scedrov. 2009. Relating state-based and process-based concurrency through linear logic (full-version). *Information and Computation* 207, 10 (2009), 1044–1077. https://doi.org/10.1016/j.ic.2008.11.006

[10] R. H. Connelly and F. Lockwood Morris. 1995. A generalisation of the trie data structure. *Mathematical Structures in Computer Science* 5, 3 (1995), 381–418. https://doi.org/10.1017/S0960129500000803

[11] Ornela Dardha. 2014. Recursive session types revisited. In *Proceedings of the 3rd Workshop on Behavioural Types (BEAT'14)*. 27–34. https://doi.org/10.4204/EPTCS.162.4

[12] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. 2017. Session types revisited. *Information and Computation* 256 (2017), 253–286. https://doi.org/10.1016/j.ic.2017.06.002

[13] Ankush Das, Farzaneh Derakhshan, and Frank Pfenning. 2019. Rast Implementation. Retrieved November 11, 2019 from https://bitbucket.org/fpfenning/rast/src/master/.

[14] Ankush Das, Henry DeYoung, Andreia Mordido, and Frank Pfenning. 2021. Nested session types. In *Programming Languages and Systems*. Lecture Notes in Computer Science, Vol. 12648. Springer, 178–206. https://doi.org/10.1007/978-3-030-72019-3_7

[15] Ankush Das, Henry DeYoung, Andreia Mordido, and Frank Pfenning. 2021. Subtyping on nested polymorphic session types. arXiv:2103.15193 [cs.PL] (2021).

[16] Ankush Das, Jan Hoffmann, and Frank Pfenning. 2018. Parallel complexity analysis with temporal session types. *Proceedings of the ACM on Programming Languages* 2, ICFP (July 2018), Article 91, 30 pages. https://doi.org/10.1145/3236786

[17] Ankush Das, Jan Hoffmann, and Frank Pfenning. 2018. Work analysis with resource-aware session types. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'18)*. ACM, New York, NY, 305–314. https://doi.org/10.1145/3209108.3209146

[18] Ankush Das and Frank Pfenning. 2020. Rast: Resource-aware session types with arithmetic refinements (system description). In *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 167)*, Zena M. Ariola (Ed.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, Article 33, 17 pages. https://doi.org/10.4230/LIPIcs.FSCD.2020.33

[19] Ankush Das and Frank Pfenning. 2020. Session types with arithmetic refinements. In *31st International Conference on Concurrency Theory (CONCUR 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 171)*, Igor Konnov and Laura Kovács (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, Article 13, 18 pages. https://doi.org/10.4230/LIPIcs.CONCUR.2020.13

[20] Ankush Das and Frank Pfenning. 2020. Verified linear session-typed concurrent programming. In *Proceedings of the 22nd International Symposium on Principles and Practice of Declarative Programming (PPDP'20)*. ACM, New York, NY, Article 7, 15 pages. https://doi.org/10.1145/3414080.3414087

[21] Farzaneh Derakhshan and Frank Pfenning. 2021. Circular proofs as session-typed processes: A local validity condition. arXiv:1908.01909 [cs.LO] (2021).

[22] Dyck. 1882. Gruppentheoretische Studien. (Mit drei lithographirten Tafeln.). *Mathematische Annalen* 20 (1882), 1–44. http://eudml.org/doc/157013.

[23] Emily P. Friedman. 1976. The inclusion problem for simple languages. *Theoretical Computer Science* 1, 4 (1976), 297–316. https://doi.org/10.1016/0304-3975(76)90074-8

[24] Peng Fu, Ekaterina Komendantskaya, Tom Schrijvers, and Andrew Pond. 2016. Proof relevant corecursive resolution. In *Functional and Logic Programming*, Oleg Kiselyov and Andy King (Eds.). Springer International Publishing, Cham, Switzerland, 126–143. https://doi.org/10.1007/978-3-319-29604-3_9

[25] Simon Gay and Malcolm Hole. 2005. Subtyping for session types in the pi calculus. *Acta Informatica* 42, 2 (Nov. 2005), 191–225. https://doi.org/10.1007/s00236-005-0177-z

[26] Simon J. Gay. 2008. Bounded polymorphism in session types. *Mathematical Structures in Computer Science* 18, 5 (2008), 895–930. https://doi.org/10.1017/S0960129508006944

[27] J. Y. Girard and Y. Lafont. 1987. Linear logic and lazy computation. In *TAPSOFT'87*, Hartmut Ehrig, Robert Kowalski, Giorgio Levi, and Ugo Montanari (Eds.). Springer, Berlin, Germany, 52–66. https://doi.org/10.1007/BFb0014972

[28] Dennis Griffith. 2016. *Polarized Substructural Session Types*. Ph.D. Dissertation. University of Illinois at Urbana–Champaign.

[29] Gopal Gupta, Ajay Bansal, Richard Min, Luke Simon, and Ajay Mallya. 2007. Coinductive logic programming and its applications. In *Logic Programming*, Véronica Dahl and Ilkka Niemelä (Eds.). Springer, Berlin, Germany, 27–44. https://doi.org/10.5555/1778180.1778186

[30] Patrick Henry and Géraud Sénizergues. 2013. LALBLC: A program testing the equivalence of DPDA's. In *Proceedings of the International Conference on Implementation and Application of Automata*. 169–180. https://doi.org/10.1007/978-3-642-39274-0_16

[31] Ralf Hinze. 2010. Generalizing generalized tries. *Journal of Functional Programming* 10, 4 (July 2010), 327–351. https://doi.org/10.1017/S0956796800003713

[32] Kohei Honda. 1993. Types for dyadic interaction. In *CONCUR'93*, Eike Best (Ed.). Springer, Berlin, Germany, 509–523. https://doi.org/10.1007/3-540-57208-2_35

[33] Petr Jančar. 2010. Short decidability proof for DPDA language equivalence via 1st order grammar bisimilarity. *CoRR* abs/1010.4760 (2010).

[34] Patricia Johann and Neil Ghani. 2009. A principled approach to programming with nested types in Haskell. *Higher-Order and Symbolic Computation* 22, 2 (June 2009), 155–189. https://doi.org/10.1007/s10990-009-9047-7

[35] Alex C. Keizer, Henning Basold, and Jorge A. Pérez. 2021. A coalgebraic view on session types and communication protocols. In *Programming Languages and Systems—30th European Symposium on Programming*, Nobuko Yoshida (Ed.). Springer, 375–403. https://doi.org/10.1007/978-3-030-72019-3_14

[36] Naoki Kobayashi. 2002. Type systems for concurrent programs. In *Formal Methods at the Crossroads*. Lecture Notes in Computer Science, Vol. 2757. Springer, 439–453. https://doi.org/10.1007/978-3-540-40007-3_26

[37] Ekaterina Komendantskaya, Dmitry Rozplokhas, and Henning Basold. 2020. The new normal: We cannot eliminate cuts in coinductive calculi, but we can explore them. *Theory and Practice of Logic Programming* 20, 6 (2020), 990–1005. https://doi.org/10.1017/S1471068420000423

[38] Allen J. Korenjak and John E. Hopcroft. 1966. Simple deterministic languages. In *Proceedings of the 7th Annual Symposium on Switching and Automata Theory (swat'66)*. IEEE, Los Alamitos, CA, 36–46. https://doi.org/10.1109/SWAT.1966.22

[39] Sam Lindley and J. Garrett Morris. 2016. Talking bananas: Structural recursion for session types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP'16)*. ACM, New York, NY, 434–447. https://doi.org/10.1145/2951913.2951921

[40] Alan Mycroft. 1984. Polymorphic type schemes and recursive definitions. In *International Symposium on Programming*, M. Paul and B. Robinet (Eds.). Springer, Berlin, Germany, 217–228. https://doi.org/10.1007/3-540-12925-1_41

[41] Chris Okasaki. 1996. *Purely Functional Data Structures*. Ph.D. Dissertation. Department of Computer Science, Carnegie Mellon University.

[42] Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. 2014. Linear logical relations and observational equivalences for session-based concurrency. *Information and Computation* 239 (2014), 254–302. https://doi.org/10.1016/j.ic.2014.08.001

[43] Frank Pfenning and Dennis Griffith. 2015. Polarized substructural session types. In *Foundations of Software Science and Computation Structures*, Andrew Pitts (Ed.). Springer, Berlin, Germany, 3–22. https://doi.org/10.1007/978-3-662-46678-0_1

[44] Benjamin C. Pierce and David N. Turner. 2000. Local type inference. *ACM Transactions on Programming Languages and Systems* 22, 1 (Jan. 2000), 1–44. https://doi.org/10.1145/345099.345100

[45] Grigore Roşu and Dorel Lucanu. 2009. Circular coinduction: A proof theoretical foundation. In *Algebra and Coalgebra in Computer Science*, Alexander Kurz, Marina Lenisa, and Andrzej Tarlecki (Eds.). Springer, Berlin, Germany, 127–144. https://doi.org/10.1007/978-3-642-03741-2_10

[46] Davide Sangiorgi. 1998. On the bisimulation proof method. *Mathematical Structures in Computer Science* 8, 5 (1998), 447–479. https://doi.org/10.1017/S0960129598002527

[47] Géraud Sénizergues. 2002. L(A)=L(B)? A simplified decidability proof. *Theoretical Computer Science* 281, 1-2 (2002), 555–608. https://doi.org/10.1016/S0304-3975(02)00027-0

[48] Marvin H. Solomon. 1978. Type definitions with parameters. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski (Eds.). ACM, New York, NY, 31–38. https://doi.org/10.1145/512760.512765

[49] Colin Stirling. 2001. Decidability of DPDA equivalence. *Theoretical Computer Science* 255, 1-2 (2001), 1–31. https://doi.org/10.1016/S0304-3975(00)00389-3

[50] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. 1994. An interaction-based language and its typing system. In *Parallel Architectures and Languages Europe*. Lecture Notes in Computer Science, Vol. 817. Springer, 398–413. https://doi.org/10.1007/3-540-58184-7_118

[51] Peter Thiemann and Vasco T. Vasconcelos. 2016. Context-free session types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP'16)*. ACM, New York, NY, 462–475. https://doi.org/10.1145/2951913.2951926

[52] Peter Thiemann and Vasco T. Vasconcelos. 2020. Label-dependent session types. *Proceedings of the ACM on Programming Languages* 4, POPL (2020), Article 67, 29 pages. https://doi.org/10.1145/3371135

[53] Philip Wadler. 2012. Propositions as sessions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'12)*. ACM, New York, NY, 273–286. https://doi.org/10.1145/2364527.2364568

Received April 2021; revised December 2021; accepted January 2022