The Different Shades of Infinite Session Types *

Simon J. Gay[1], Diogo Poças[2](✉), and Vasco T. Vasconcelos[2]

[1] School of Computing Science, University of Glasgow, UK
simon.gay@glasgow.ac.uk
[2] LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
{dmpocas,vmvasconcelos}@ciencias.ulisboa.pt

Abstract. Many type systems include infinite types. In session type systems, infinite types are important because they specify communication protocols that are unbounded in time. Usually infinite session types are introduced as simple finite-state expressions $\mathsf{rec}\, X.T$ or by non-parametric equational definitions $X \doteq T$. Alternatively, some systems of label- or value-dependent session types go beyond simple recursive types. However, leaving dependent types aside, there is a much richer world of infinite session types, ranging through various forms of parametric equational definitions, to arbitrary infinite types in a coinductively defined space. We study infinite session types across a spectrum of shades of grey on the way to the bright light of general infinite types. We identify four points on the spectrum, characterised by different styles of equational definitions, and show that they form a strict hierarchy by establishing bidirectional correspondences with classes of automata: finite-state, 1-counter, pushdown and 2-counter. This allows us to establish decidability and undecidability results for type formation, type equivalence and duality in each class of types. We also consider previous work on context-free session types (and extend it to higher-order) and nested session types, and locate them on our spectrum of infinite types.

Keywords: Infinite types · Recursive types · Session types · Automata and formal language theory

1 Introduction

Session types [19,20,23,38] are an established approach to specifying communication protocols, so that protocol implementations can be verified by static typechecking or dynamic monitoring. The simplest protocols are finite: for example, ?int.!bool.end describes a protocol in which an integer is received, then a boolean is sent, and that's all. Most systems of session types, however, include

*Supported by EPSRC EP/T014628/1 "Session Types for Reliable Distributed Systems", by FCT PTDC/CCI-CIF/6453/2020 "Safe Concurrent Programming with Session Types", and by the LASIGE Research Unit UIDB/00408/2020 and UIDP/00408/2020. A full version is available at https://arxiv.org/abs/2201.08275 [16].

P. Bouyer and L. Schröder (Eds.): FoSSaCS 2022, LNCS 13242, pp. 347–367, 2022.
https://doi.org/10.1007/978-3-030-99253-8_18

equi-recursive types for greater expressivity. A type that endlessly repeats the simple send-receive protocol is X such that $X \doteq ?\mathsf{int}.!\mathsf{bool}.X$, which can also be specified by $\mathsf{rec}\, X.?\mathsf{int}.!\mathsf{bool}.X$. More realistic examples usually combine recursion and choice, as in Y such that $Y \doteq \&\{\mathrm{go}\colon ?\mathsf{int}.!\mathsf{bool}.Y, \mathrm{quit}\colon \mathsf{end}\}$ which offers a choice between go and quit operations, each with its own protocol. A natural observation is that session types look like finite-state automata, but some systems from the literature go beyond the finite-state format: for example, context-free session types [39] and nested session types [9,10], as well as label-dependent session types [40] and value-dependent session types [41].

Even without introducing dependent types, a range of definitional formats can be considered for session types, presumably with varying degrees of expressivity, but they have never been systematically studied. That is the aim of the present paper. We consider various forms of parameterised equational definitions, illustrated by six running examples. Because our formal system only has one base type, the terminated channel type end, the running examples simply use end (or skip for context-free session types) as a representative basic message type that could otherwise be bool or int.

Our study of classes of infinite types should be generally applicable; we make it concrete by concentrating on session types where (potential) infinite types occur naturally. For the sake of uniformity, all our non-finite session types are introduced by equations, rather than, say, rec-types. Equations may be further parameterized, thus accounting for types that go beyond recursive types. The examples below illustrate the different kinds of parameterized equations we use.

Example 1 (No parameters). Type T_{loop} is X with equation $X \doteq !\mathsf{end}.X$. Intuitively $T_{\mathrm{loop}} = !\mathsf{end}.!\mathsf{end}\ldots$ continuously outputs values of type end.

Example 2 (One natural number parameter). Assuming z and s as the natural number constructors and N as a variable over natural numbers, type T_{counter} is $X\langle \mathsf{z}\rangle$ with equations

$$X\langle \mathsf{z}\rangle \doteq \&\{\mathrm{inc}\colon X\langle \mathsf{s}\,\mathsf{z}\rangle, \mathrm{dump}\colon Y\langle \mathsf{z}\rangle\} \qquad Y\langle \mathsf{z}\rangle \doteq \mathsf{end}$$
$$X\langle \mathsf{s}\,N\rangle \doteq \&\{\mathrm{inc}\colon X\langle \mathsf{s}\,\mathsf{s}\,N\rangle, \mathrm{dump}\colon Y\langle \mathsf{s}\,N\rangle\} \qquad Y\langle \mathsf{s}\,N\rangle \doteq !\mathsf{end}.Y\langle N\rangle$$

A sequence of n inc operations followed by a dump triggers a reply of n end output messages.

Example 3 (Context-free types). With type skip used either to finish a session or to move to the next operation, type T_{tree} is X with equation

$$X \doteq \&\{\mathrm{leaf}\colon \mathsf{skip}, \mathrm{node}\colon X; ?\mathsf{skip}; X\}$$

The leaf choice terminates the reception of a binary tree of skip values and the node choice triggers the reception of a (left) tree, followed by ?skip (root), followed by a (right) tree. Even though the development in the rest of the paper considers higher-order types (where messages may convey arbitrary types rather than skip alone), for simplicity our example is first-order.

Example 4 (One list parameter). Assuming σ and τ as symbols and S as a variable over sequences of symbols (with ε the empty sequence), type T_{meta} is $X\langle\varepsilon\rangle$ with equations

$$\begin{aligned} X\langle\varepsilon\rangle &\doteq \&\{\mathsf{addOut}\colon X\langle\sigma\rangle, \mathsf{addIn}\colon X\langle\tau\rangle\} \\ X\langle\sigma S\rangle &\doteq \&\{\mathsf{addOut}\colon X\langle\sigma\sigma S\rangle, \mathsf{addIn}\colon X\langle\tau\sigma S\rangle, \mathsf{pop}\colon \mathsf{!end}.X\langle S\rangle\} \\ X\langle\tau S\rangle &\doteq \&\{\mathsf{addOut}\colon X\langle\sigma\tau S\rangle, \mathsf{addIn}\colon X\langle\tau\tau S\rangle, \mathsf{pop}\colon \mathsf{?end}.X\langle S\rangle\} \end{aligned}$$

Type T_{meta} records simple protocols composed of !end and ?end messages. Symbol σ in a parameter to a type identifier X denotes an output message and symbol τ an input message. The protocol behaves as a stack with two distinct push operations (addOut and addIn). The symbol (σ or τ) at top of the stack determines whether a pop operation triggers !end or ?end, respectively.

Example 5 (Nested types). Taking α as a variable over types, type T_{nest} is X_ε with equations

$$\begin{aligned} X_\varepsilon &\doteq \&\{\mathsf{addOut}\colon X_{\text{out}}\langle X_\varepsilon\rangle, \mathsf{addIn}\colon X_{\text{in}}\langle X_\varepsilon\rangle\} \\ X_{\text{out}}\langle\alpha\rangle &\doteq \&\{\mathsf{addOut}\colon X_{\text{out}}\langle X_{\text{out}}\langle\alpha\rangle\rangle, \mathsf{addIn}\colon X_{\text{in}}\langle X_{\text{out}}\langle\alpha\rangle\rangle, \mathsf{pop}\colon \mathsf{!end}.\alpha\} \\ X_{\text{in}}\langle\alpha\rangle &\doteq \&\{\mathsf{addOut}\colon X_{\text{out}}\langle X_{\text{in}}\langle\alpha\rangle\rangle, \mathsf{addIn}\colon X_{\text{in}}\langle X_{\text{in}}\langle\alpha\rangle\rangle, \mathsf{pop}\colon \mathsf{?end}.\alpha\} \end{aligned}$$

Type identifiers such as $X_\varepsilon, X_{\text{out}}, X_{\text{in}}$ take an arbitrary but fixed number of arguments. Type T_{nest} behaves as T_{meta} in Example 4. The alignment should be clear if we take, e.g. $X_{\text{out}}\langle X_{\text{in}}\langle\alpha\rangle\rangle$ for $X\langle\sigma\tau S\rangle$, with σ denoting output and τ denoting input. Type identifiers X_{out} and X_{in} play the roles of stack symbols (symbols at the top of the stack, σ or τ); type variable α denotes the lower part of the stack (S in Example 4).

Example 6 (Two natural number parameters). Type T_{iter} is $X\langle\mathsf{z},\mathsf{z}\rangle$ with

$$\begin{aligned} X\langle\mathsf{z},N'\rangle &\doteq \mathsf{?end}.Y\langle\mathsf{z},\mathsf{s}\,N'\rangle & Y\langle N,\mathsf{z}\rangle &\doteq X\langle N,\mathsf{z}\rangle \\ X\langle\mathsf{s}\,N,N'\rangle &\doteq \mathsf{!end}.X\langle N,\mathsf{s}\,N'\rangle & Y\langle N,\mathsf{s}\,N'\rangle &\doteq Y\langle\mathsf{s}\,N,N'\rangle \end{aligned}$$

Informally, writing $\mathsf{!end}^n$ for a sequence of n output end messages, these definitions give $T_{\text{iter}} = \mathsf{?end}.\mathsf{!end}^1.\mathsf{?end}.\mathsf{!end}^2.\mathsf{?end}.\mathsf{!end}^3\ldots$

It is intuitively clear that Examples 2 and 4 to 6 cannot be expressed without parameters. It is perhaps less clear that each definitional style in Examples 1, 2, 4 and 6 is strictly more expressive than the previous one. This is the main result of the paper. We establish a hierarchy from finite session types all the way up to non-computable types that have no representation at all. The latter certainly exist, because for every infinite binary expansion of a real number between zero and one there is a session type derived by mapping 0 to send and 1 to receive — for cardinality reasons, almost all of these types are non-computable.

Our methodology is to develop the connection between session types and automata, in particular between progressively more expressive definitional styles of types and progressively more powerful classes of automata. We also consider

the formal language class corresponding to each class of automata, and the decidability of important properties such as contractiveness, type formation, type equivalence and type duality. Our results are summarised in the table below, establishing a hierarchy of session types in parallel to the Chomsky hierarchy of languages, where by a 1-counter language, we mean a language accepted by a (deterministic) 1-counter automaton and where DCFL abbreviates deterministic context-free languages. The final row of the table emphasises that it is impossible to give an explicit example of a non-computable type or to even state the decision problems.

Context-free and 1-counter types are incomparable. Essentially, both models lie between levels 2 and 3 of the Chomsky hierarchy and correspond to different restrictions of deterministic pushdown automata. Context-free types correspond to constraining automata with a single state, whereas 1-counter types correspond to constraining the stack to have a single symbol.

Type class	Example	Contractiveness	Type duality / equivalence	Language model
Finite	!end.end	Polytime	Polytime	Finite languages
Recursive	T_{loop}	Polytime	Polytime	Regular languages
1-counter	T_{counter}	Polytime	Polytime	1-counter languages
HO context-free	T_{tree}	Polytime	Decidable	Open[3]
Pushdown	T_{meta}	Polytime	Decidable	DCFL
Nested	T_{nest}	Polytime	Decidable	DCFL
2-counter	T_{iter}	Undecidable	Undecidable	Decidable languages
Non-computable	—	—	—	General languages

Our main contributions can be summarized as follows.

- We propose three novel formal systems for representing session types (1-counter, pushdown, 2-counter), show that they are strictly more expressive than recursive session types, and that each system is strictly more expressive than the previous one (Theorem 1).
- We show that nested session types [9] are equivalent to pushdown session types (Theorem 1).
- We introduce higher-order context-free session types and show that they stand between recursive and pushdown types, strictly (Theorem 1).
- We characterize each of the novel session types in our paper by a corresponding class in the Chomsky hierarchy of languages. Notably, we show that each model captures precisely the power of the corresponding class of automata (Theorem 2). This is in contrast with the results of Das et al. [9], who only show (in one direction) that nested session types can be simulated by deterministic pushdown automata.
- We prove that type formation, type equivalence and type duality are decidable up to pushdown session types (Theorem 3), but undecidable for 2-

[3]Possibly languages accepted by a single-state pushdown automata with empty stack acceptance.

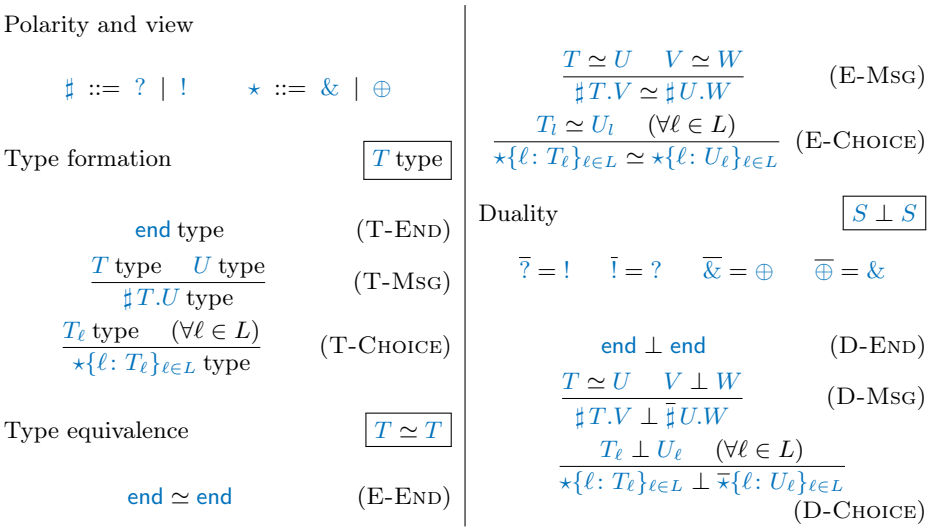


Fig. 1. Finite and infinite types.

counter session types (Theorem 4). This implies that equivalence for higher-order context-free session types is decidable. The decidability results are not entirely unexpected, given that type equivalence for nested session types was recently shown to be decidable [9], and that these are equivalent to pushdown types. However, our proofs are independent of Das et al. [9].

Organization of the paper In Section 2 we introduce the various classes of session types. In Section 3 we explain how to associate to each given type a labelled infinite tree, as well as a set which we call the language of traces of that type. We also present our results on the strict hierarchy of types and state how previously studied classes of types fit into this hierarchy (Theorem 1). In Section 4 we describe how to convert a type into an automaton accepting its traces. In Section 5 we travel in the converse direction, i.e., from an automata into the corresponding type, and present a characterisation theorem of the different types in our hierarchy (Theorem 2). We then present our main algorithmic results: type formation, type equivalence and type duality are all decidable up to pushdown types (Theorem 3), and undecidable for 2-counter types (Theorem 4). Due to space constraints, all proofs and additional details can be found in the extended version of our paper at arXiv [16].

2 Shades of types

The finite world Finite types are in Fig. 1. The syntax of types is introduced via formation rules, paving the way for infinite types. Session types comprise the terminated type end, the input type $?T.U$ (input a value of type T and

Type contractivity (*ind.*) $\boxed{T \text{ contr}}$

$$\mathsf{end} \text{ contr} \quad \text{(C-End)}$$

$$\sharp T.U \text{ contr} \quad \text{(C-Msg)}$$

$$\star\{\ell\colon T_\ell\}_{\ell\in L} \text{ contr} \quad \text{(C-Choice)}$$

$$\frac{X \doteq T \qquad T \text{ contr}}{X \text{ contr}} \quad \text{(C-Id)}$$

New formation rules (*coind.*) $\boxed{T \text{ type}}$

$$\frac{X \doteq T \qquad T \text{ contr} \qquad T \text{ type}}{X \text{ type}} \quad \text{(T-Id)}$$

New equivalence rules (*coind.*) $\boxed{T \simeq T}$

$$\frac{X \doteq U \qquad U \text{ contr} \qquad U \simeq T}{X \simeq T} \quad \text{(E-ConsL)}$$

$$\frac{X \doteq U \qquad U \text{ contr} \qquad T \simeq U}{T \simeq X} \quad \text{(E-ConsR)}$$

New duality rules (*coind.*) $\boxed{T \perp T}$

$$\frac{X \doteq U \qquad U \text{ contr} \qquad U \perp T}{X \perp T} \quad \text{(D-IdL)}$$

$$\frac{X \doteq U \qquad U \text{ contr} \qquad T \perp U}{T \perp X} \quad \text{(D-IdR)}$$

Fig. 2. Recursive types. Extends Fig. 1.

continue as U), the output type $!T.U$ (output a value of type T and continue as U), external choice $\&\{\ell\colon T_\ell\}_{\ell\in L}$ (receive a label $k \in L$ and continue as T_k) and internal choice $\oplus\{\ell\colon T_\ell\}_{\ell\in L}$ (select a label $k \in L$ and continue as T_k). To avoid repeating similar rules, we use the symbol $\sharp$ to denote either $?$ or $!$, and the symbol $\star$ to denote either $\&$ or $\oplus$. At this point type equivalence is essentially syntactic equality, but the rule format allows for seamless extensions to infinite settings. Types, type equivalence and duality are all standard [15,20,44]. Note that rule D-Msg defines duality with respect to type equivalence: $!T.V$ and $?U.W$ are dual types iff the type being exchanged is the same ($T \simeq U$) and the continuations are dual ($V \perp W$).

For finite types all judgements in Fig. 1 are interpreted *inductively*. For example, we can show that $!(?\mathsf{end}.\mathsf{end}).!\mathsf{end}.\mathsf{end}$ is a type by exhibiting a finite derivation ending with this judgement.

The recursive world Recursive types suggest the first glimpse of infinity. The details are in Fig. 2. Recursion is given via equations, rather than μ-types for example, for easier extension. Towards this end, we introduce type identifiers X and equations of the form $X \doteq T$. The set of type identifiers is finite. We further assume at most one equation for each type, so that there are finitely many type equations. Every valid type T is required to be contractive, that is T contr. Contractiveness ensures that types reveal a type constructor after finitely many unfolds, and excludes undesirable cycles that don't describe any behaviour, e.g. cycles of the form $\{X \doteq Y, Y \doteq Z, Z \doteq X\}$. Contractiveness is inductive: we look for finite derivations for T contr judgements. A coinductive interpretation of the rules would allow to conclude X contr given an equation $X \doteq X$. In contrast, type formation, type equivalence and duality are now interpreted *coinductively*.

For example, no finite derivation would allow showing that T_{loop} type. Instead we proceed by showing that set $\{\mathsf{end}, !\mathsf{end}.X, X\}$ is *backward closed* [34] for the rules for T type in Fig. 2, given that $!\mathsf{end}.X$, the right-hand side of the equation for X, is contractive.

Natural numbers

$$n ::= \mathsf{z} \mid \mathsf{s}\,n$$

New contractivity rules (*ind.*) $\boxed{T \text{ contr}}$

$$\frac{X\langle \mathsf{z}\rangle \doteq T \quad T \text{ contr}}{X\langle \mathsf{z}\rangle \text{ contr}} \text{ (C-z)} \qquad \frac{X\langle \mathsf{s}\,N\rangle \doteq T \quad T[n/N] \text{ contr}}{X\langle \mathsf{s}\,n\rangle \text{ contr}} \text{ (C-s)}$$

New formation rule (*coind.*) $\boxed{T \text{ type}}$

$$\frac{X\langle \mathsf{z}\rangle \doteq T \quad T \text{ contr} \quad T \text{ type}}{X\langle \mathsf{z}\rangle \text{ type}} \text{ (T-z)} \qquad \frac{X\langle \mathsf{s}\,N\rangle \doteq T \quad T[n/N] \text{ contr} \quad T[n/N] \text{ type}}{X\langle \mathsf{s}\,n\rangle \text{ type}} \text{ (T-s)}$$

New equivalence rules (*coind.*) $\boxed{T \simeq T}$

$$\frac{X\langle \mathsf{z}\rangle \doteq U \quad U \text{ contr} \quad U \simeq T}{X\langle \mathsf{z}\rangle \simeq T} \text{ (E-zL)} \qquad \frac{X\langle \mathsf{s}\,N\rangle \doteq U \quad U[n/N] \text{ contr} \quad U[n/N] \simeq T}{X\langle \mathsf{s}\,n\rangle \simeq T} \text{ (E-sL)}$$

Fig. 3. 1-counter types. Extends Fig. 2; removes X; adds $X\langle n\rangle$. Right versions of rules E-zL and E-sL omitted. New rules for duality obtained from those for equivalence by replacing $\simeq$ by $\perp$.

The 1-counter world The next step takes us to equations parameterised on natural numbers. The details are in Fig. 3. Natural numbers are built from the nullary constructor z and the unary constructor s. We discuss the changes from the recursive world in Fig. 2. Given a variable N on natural numbers, to each type identifier X we associate at most two equations, $X\langle \mathsf{z}\rangle \doteq T$ and $X\langle \mathsf{s}\,N\rangle \doteq U$. The rules for recursive types are naturally adapted to 1-counter types. Here again, type formation requires a suitable notion of contractiveness to exclude cycles of equations that never reach a type identifier, e.g. cycles of the form $\{X\langle \mathsf{s}\,N\rangle \doteq Y\langle \mathsf{s}\,\mathsf{s}\,N\rangle, Y\langle \mathsf{s}\,N\rangle \doteq X\langle N\rangle\}$. The right-hand-side of an equation $X\langle \mathsf{s}\,N\rangle \doteq T$ is not necessarily a type for it may contain natural number variables (N in particular). However, if n is a natural number, then $T[n/N]$ (that is, T with occurrences of N replaced by n) should be a type (cf. rule T-s). Again, to prove that T_{counter} type, we show backward closure of the set $\{X\langle n\rangle, Y\langle n\rangle, \mathsf{end}, \mathsf{!end}.Y\langle n\rangle, \&\{\text{inc}\colon X\langle \mathsf{s}\,n\rangle, \text{dump}\colon Y\langle n\rangle\} \mid n \text{ nat}\}$ for the type formation rules.

Higher-order context-free session types A little detour takes us to context-free session types, proposed by Thiemann and Vasconcelos [39] (see also Almeida et al. [1]). Here we follow the distilled presentation of Almeida et al. [2], extending types to the higher-order setting (that is, allowing $?T$ and $!T$ for an arbitrary type T instead of just basic type skip).

The pushdown world The next extension replaces natural numbers by finite sequences s of symbols σ taken from a given stack alphabet. The details are in Fig. 4. We use ε to denote the empty sequence. The extension from 1-counter is straightforward. Parameters to type identifiers are now sequences of symbols, rather than natural numbers; all the rest remains the same. Once again, to show that T_{meta} type, we proceed coinductively.

Strings

$$s ::= \varepsilon \mid \sigma s$$

New contractive rules (*ind.*) $\boxed{T \text{ contr}}$

$$\frac{X\langle\varepsilon\rangle \doteq T \quad T \text{ contr}}{X\langle\varepsilon\rangle \text{ contr}} \text{ (C-z)} \qquad \frac{X\langle\sigma S\rangle \doteq T \quad T[s/S] \text{ contr}}{X\langle\sigma s\rangle \text{ contr}} \text{ (C-s)}$$

New formation rules (*coind.*) $\boxed{T \text{ type}}$

$$\frac{X\langle\varepsilon\rangle \doteq T \quad T \text{ contr} \quad T \text{ type}}{X\langle\varepsilon\rangle \text{ type}} \text{ (T-z)} \qquad \frac{X\langle\sigma S\rangle \doteq T \quad T[s/S] \text{ contr} \quad T[s/S] \text{ type}}{X\langle\sigma s\rangle \text{ type}} \text{ (T-s)}$$

New equivalence rules (*coind.*) $\boxed{T \simeq T}$

$$\frac{X\langle\varepsilon\rangle \doteq U \quad U \text{ contr} \quad U \simeq T}{X\langle\varepsilon\rangle \simeq T} \text{ (E-zL)} \qquad \frac{X\langle\sigma S\rangle \doteq U \quad U[s/S] \text{ contr} \quad U[s/S] \simeq T}{X\langle\sigma s\rangle \simeq T} \text{ (E-sL)}$$

Fig. 4. Pushdown types. Extends Fig. 2; removes X; adds $X\langle s\rangle$. Right versions of rules E-zL and E-sL omitted. For duality, proceed as in Fig. 3.

Nested session types A class of types that turns out to be equivalent to pushdown types was recently proposed by Das et al. [9]. The main idea is to have type identifiers that are applied not to natural numbers or to sequences of symbols but to types themselves, and to let type identifiers take a variable (but fixed) number of parameters.

The 2-counter world 2-counter types extend the 1-counter types by introducing equations parameterised on two natural numbers, rather than one. The new rules are a straightforward adaptation of those in Fig. 3 for 1-counter types and are thus omitted. To show that T_{iter} type, we proceed coinductively.

The infinite world The final destination takes us to arbitrary, coinductive, infinite types. The details are in Fig. 1, except that all judgements not explicitly marked are taken coinductively. No equations (of any sort) are needed, just plain infinite types. We also allow choices with an infinite number of branches.

Infinite types arise by interpreting the syntax rules coinductively, which gives rise to potentially infinite chains of interactions. The structure of these arbitrary, coinductively defined, infinite types does not need to follow any pattern (e.g. it does not need to repeat itself), and arguably, the best way to think about these objects are as labelled infinite trees (Section 3). Such objects do not have in general a finite representation (or finite encoding), which can be shown by a simple cardinality argument. Hence the need for finding suitable subclasses of infinite types that can be represented and can be used in practice.

We can think of a type in two possible ways: as (one of) its representation(s), which is great for practical purposes as we can reason about types by reasoning about their representations; or as the underlying, possibly infinite, coinductive object which is being represented, which is suitable for developing a theory of types, in particular for comparing different models with one another.

3 Types, trees and traces

It should be clear that the constructions defined in Section 2 form some sort of type hierarchy; this section studies the hierarchy. In any case, every type lives in the largest universe; that of arbitrary, coinductively defined, infinite types.

To each type one can associate a labelled infinite tree [14,32]. This tree can in turn be expressed by the language of words encoding its paths. Let $\mathbb{L}$ be the set of labels used in choice types. Following Pierce [32, Definition 21.2.1], a tree is a partial function $t \in (\{\mathsf{d},\mathsf{c}\} \cup \mathbb{L})^* \to \{\mathsf{end}, ?, !, \&_L, \oplus_L \mid L \subseteq \mathbb{L}\}$ subject to the following constraints (σ ranges over symbols and π over strings of symbols):

- $t(\varepsilon)$ is defined;
- if $t(\pi\sigma)$ is defined, then $t(\pi)$ is defined;
- if $t(\pi) = ?$ or $t(\pi) = !$, then $t(\pi\sigma)$ is defined for $\sigma \in \{\mathsf{d},\mathsf{c}\}$ and undefined for all other σ;
- if $t(\pi) = \&_L$ or $t(\pi) = \oplus_L$, then $t(\pi\sigma)$ is defined for $\sigma \in L$ and undefined for all other σ;
- if $t(\pi) = \mathsf{end}$, then $t(\pi\sigma)$ is undefined for all σ.

The labels d and c are abbreviations for *data* and *continuation*, corresponding to the two components of session types for messages.

If all sets L in a tree are finite, the tree is finitely branching. The tree generated by a (finite or infinite) type is coinductively defined as follows.

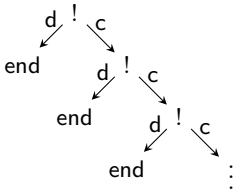
$$\begin{aligned}
\mathrm{treeof}(\sharp\, T_{\mathsf{d}}.T_{\mathsf{c}})(\varepsilon) &= \sharp & \mathrm{treeof}(\star\{\ell\colon T_\ell\}_{\ell\in L})(\varepsilon) &= \star_L \\
\mathrm{treeof}(\sharp\, T_{\mathsf{d}}.T_{\mathsf{c}})(\mathsf{d}\pi) &= \mathrm{treeof}(T_{\mathsf{d}})(\pi) & \mathrm{treeof}(\star\{\ell\colon T_\ell\}_{\ell\in L})(\ell\pi) &= \mathrm{treeof}(T_\ell)(\pi) \\
\mathrm{treeof}(\sharp\, T_{\mathsf{d}}.T_{\mathsf{c}})(\mathsf{c}\pi) &= \mathrm{treeof}(T_{\mathsf{c}})(\pi) & \mathrm{treeof}(\mathsf{end})(\varepsilon) &= \mathsf{end}
\end{aligned}$$

A *path* in a tree t is a word obtained by combining the symbols in the domain and the range of t. Given a symbol $\sigma \in \{?, !, \&_L, \oplus_L \mid L \subseteq \mathbb{L}\}$ in the codomain of t (but different from end), and a symbol $\tau \in \{\mathsf{d},\mathsf{c}\} \cup \mathbb{L}$, let $\langle\sigma,\tau\rangle$ denote the combination of both symbols, viewed as a letter over the alphabet $\{?, !, \&_L, \oplus_L \mid L \subseteq \mathbb{L}\} \times (\{\mathsf{d},\mathsf{c}\} \cup \mathbb{L})$. For simplicity in exposition, we often drop the angular brackets and the subscript L on the label set, and write, for example, $?\mathsf{c}$ instead of $\langle ?, \mathsf{c}\rangle$, $\oplus l$ instead of $\langle \oplus_L, l\rangle$, etc.

Given a string π in the domain of a tree t, we can define the word $\mathrm{path}_t(\pi)$ recursively as $\mathrm{path}_t(\varepsilon) = \varepsilon$ and $\mathrm{path}_t(\pi\tau) = \mathrm{path}_t(\pi) \cdot \langle t(\pi), \tau\rangle$. We say that a string π *is terminal wrt to* t if $t(\pi) = \mathsf{end}$. For terminal strings, we can further define $\mathrm{dpath}_t(\pi) = \mathrm{path}_t(\pi) \cdot \mathsf{end}$.

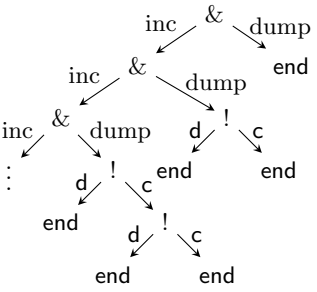
Finally, we can define the language of (the paths in) a tree t as the set $\{\mathrm{path}_t(\pi) \mid \pi \in \mathrm{dom}(t)\} \cup \{\mathrm{dpath}_t(\pi) \mid \pi \in \mathrm{dom}(t), \pi \text{ is terminal wrt } t\}$. The language of (the traces of) a type T, denoted by $\mathcal{L}(T)$, is the language of $\mathrm{treeof}(T)$. Note that the traces of types are defined over the following alphabet.

$$\Sigma = \{?, !, \&_L, \oplus_L \mid L \subseteq \mathbb{L}\} \times (\{\mathsf{d},\mathsf{c}\} \cup \mathbb{L}) \cup \{\mathsf{end}\} \tag{1}$$



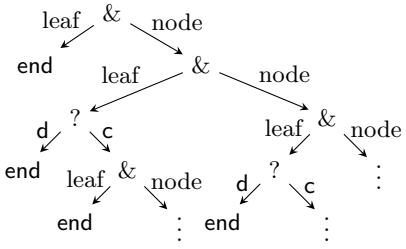
$$\mathcal{L}(T_{\text{loop}}) = \{\varepsilon,\ \ !\mathsf{d},\ \ !\mathsf{d}\cdot\mathsf{end},\ \ !\mathsf{c},\ \ !\mathsf{c}\cdot!\mathsf{d},\ \ !\mathsf{c}\cdot!\mathsf{d}\cdot\mathsf{end},\ \ !\mathsf{c}\cdot!\mathsf{c},\ \ !\mathsf{c}\cdot!\mathsf{c}\cdot!\mathsf{d},\ \ !\mathsf{c}\cdot!\mathsf{c}\cdot!\mathsf{d}\cdot\mathsf{end},\ \ !\mathsf{c}\cdot!\mathsf{c}\cdot!\mathsf{c},\ \ \ldots\}$$

Fig. 5. The tree and the language of type T_{loop}.



$$\mathcal{L}(T_{\text{counter}}) = \{\varepsilon,\ \ \&\text{inc},\ \ \&\text{dump},\ \ \&\text{dump}\cdot\mathsf{end},\ \ \&\text{inc}\cdot\&\text{inc},\ \ \&\text{inc}\cdot\&\text{dump},\ \ \&\text{inc}\cdot\&\text{dump}\cdot!\mathsf{d},\ \ \&\text{inc}\cdot\&\text{dump}\cdot!\mathsf{d}\cdot\mathsf{end},\ \ \&\text{inc}\cdot\&\text{dump}\cdot!\mathsf{c},\ \ \&\text{inc}\cdot\&\text{dump}\cdot!\mathsf{c}\cdot\mathsf{end},\ \ \&\text{inc}\cdot\&\text{inc}\cdot\&\text{inc},\ \ \&\text{inc}\cdot\&\text{inc}\cdot\&\text{dump},\ \ \&\text{inc}\cdot\&\text{inc}\cdot\&\text{dump}\cdot!\mathsf{d},\ \ \ldots\}$$

Fig. 6. The tree and the language of type T_{counter}.



$$\mathcal{L}(T_{\text{tree}}) = \{\varepsilon,\ \ \&\text{leaf},\ \ \&\text{leaf}\cdot\mathsf{end},\ \ \&\text{node},\ \ \&\text{node}\cdot\&\text{leaf},\ \ \&\text{node}\cdot\&\text{node},\ \ \&\text{node}\cdot\&\text{leaf}\cdot?\mathsf{d},\ \ \&\text{node}\cdot\&\text{leaf}\cdot?\mathsf{d}\cdot\mathsf{end},\ \ \&\text{node}\cdot\&\text{leaf}\cdot?\mathsf{c},\ \ \&\text{node}\cdot\&\text{node}\cdot\&\text{leaf},\ \ \&\text{node}\cdot\&\text{node}\cdot\&\text{node},\ \ \ldots\}$$

Fig. 7. The tree and the language of type T_{tree}.

Figure 5 depicts (a finite fragment of) the tree corresponding to treeof(T_{loop}) (Example 1) and (some of the words in) its language $\mathcal{L}(T_{\text{loop}})$. Type T_{counter} (Example 2) describes an interaction that keeps track of a counter. Finite fragments of the corresponding tree and language are depicted in Fig. 6. Type T_{tree} (Example 3) describes the reception of a binary tree of end values. Finite fragments of the corresponding tree and language are depicted in Fig. 7.

In the above examples, the language $\mathcal{L}(T)$ is closed under prefixes. This holds for a general type T, since elements of $\mathcal{L}(T)$ correspond to paths in treeof(T).

Proposition 1. *If $w \in \mathcal{L}(T)$ and u is a prefix of w then $u \in \mathcal{L}(T)$.*

Another immediate observation is that treeof (resp. $\mathcal{L}$) is an embedding from the class of all types to the class of all trees (resp. all languages).

Proposition 2. *Let T and U be two types. The following are equivalent: a) $T \simeq U$; b) $\mathrm{treeof}(T) = \mathrm{treeof}(U)$; c) $\mathcal{L}(T) = \mathcal{L}(U)$.*

Proposition 2 tells us that two types are equivalent iff they have the same traces. In general, trace equivalence is a notion weaker than bisimulation [34]. However, both notions coincide for deterministic transition systems. The syntax of (infinite) session types is in fact deterministic (e.g. given a label ℓ for a choice, there can only be one type that continues from $\&\ell$), which explains our result.

Section 2 introduces eight classes of types. We now distinguish them by means of subscripts: finite types (T $\mathrm{type}_{\mathsf{f}}$, Fig. 1), recursive types (T $\mathrm{type}_{\mathsf{r}}$, Fig. 2), 1-counter types (T type_{1}, Fig. 3), context-free types (T $\mathrm{type}_{\mathsf{c}}$), pushdown types (T $\mathrm{type}_{\mathsf{p}}$, Fig. 4), nested types (T $\mathrm{type}_{\mathsf{n}}$), 2-counter types (T type_{2}) and coinductive, infinite types (T type_{∞}, Fig. 1 with rules interpreted coinductively). To each class of types we introduce the corresponding class of languages. For example, $\mathbb{T}_{\mathsf{r}}$ is the set $\{\mathcal{L}(T) \mid T \ \mathrm{type}_{\mathsf{r}}\}$. The strict hierarchy result is as follows:

$$\mathbb{T}_{\mathsf{f}} \subsetneq \mathbb{T}_{\mathsf{r}} \subsetneq \mathbb{T}_{1} \subsetneq \mathbb{T}_{\mathsf{p}} \subsetneq \mathbb{T}_{2} \subsetneq \mathbb{T}_{\infty}$$

We remark that the last step in the chain of strict inclusions is obtained by a cardinality argument, since the set $\mathbb{T}_{\infty}$ is uncountable. This shows an even stronger statement: for any finite representation system (including the systems $\mathbb{T}_{\mathsf{f}}$ to $\mathbb{T}_{2}$, as well as $\mathbb{T}_{\mathsf{c}}$ and $\mathbb{T}_{\mathsf{n}}$), there is an *infinite, uncountable* set of types that cannot be represented by that system.

We now turn our attention to nested types (T $\mathrm{type}_{\mathsf{n}}$), which turn out to be equivalent to pushdown types, and further establish equivalent sub-hierarchies inside both classes, parameterised by the 'complexity' of the corresponding representations. For pushdown session types, a natural measure of complexity is the number of type identifiers required to represent a given type. This number can be arbitrarily large, but always finite. For a given $n \in \mathbb{N}$, we let $\mathbb{T}^{n}_{\mathsf{p}}$ denote the subset corresponding to those types that can be represented with at most n type identifiers. When $n = 0$, there are no identifiers, and we can only represent finite types. As n increases, so does the expressivity of our constructions, and we have the infinite chain of inclusions[4]

$$\mathbb{T}_{\mathsf{f}} = \mathbb{T}^{0}_{\mathsf{p}} \subsetneq \mathbb{T}^{1}_{\mathsf{p}} \subseteq \mathbb{T}^{2}_{\mathsf{p}} \subseteq \cdots \subseteq \mathbb{T}_{\mathsf{p}}.$$

Similarly, for nested session types we can define a hierarchy by looking at the arities of the type identifiers used. For a given $n \in \mathbb{N}$, we let $\mathbb{T}^{n}_{\mathsf{n}}$ denote the subset corresponding to the nested session types whose type identifiers have arity at most n. When $n = 0$ all type identifiers are constant, and we recover the class of recursive types. As n increases, so does the expressivity, and we also have an infinite chain of inclusions[4]

$$\mathbb{T}_{\mathsf{r}} = \mathbb{T}^{0}_{\mathsf{n}} \subsetneq \mathbb{T}^{1}_{\mathsf{n}} \subseteq \mathbb{T}^{2}_{\mathsf{n}} \subseteq \cdots \subseteq \mathbb{T}_{\mathsf{n}}.$$

[4] Although not proven, we conjecture that all inclusions are strict.

It turns out that these hierarchies are one and the same (with the exception of the bottom level), so that we have [4]

$$\mathbb{T}_{\mathsf{f}} = \mathbb{T}^0_{\mathsf{p}} \subsetneq \mathbb{T}_{\mathsf{r}} = \mathbb{T}^0_{\mathsf{n}} \subsetneq \mathbb{T}^1_{\mathsf{p}} = \mathbb{T}^1_{\mathsf{n}} \subseteq \mathbb{T}^2_{\mathsf{p}} = \mathbb{T}^2_{\mathsf{n}} \subseteq \cdots \subseteq \mathbb{T}_{\mathsf{p}} = \mathbb{T}_{\mathsf{n}}.$$

Higher-order context-free types (denoted by $\mathbb{T}_{\mathsf{c}}$) lie between levels 0 and 1 in the sub-hierarchies above, i.e., they can represent recursive types, and can be represented by pushdown session types using at most one type identifier, or equivalently, by nested session types with either constant or unary type identifiers, so that we have

$$\mathbb{T}_{\mathsf{r}} \subsetneq \mathbb{T}_{\mathsf{c}} \subsetneq \mathbb{T}^1_{\mathsf{p}} = \mathbb{T}^1_{\mathsf{n}}.$$

We have a stronger observation than the inclusion $\mathbb{T}_{\mathsf{c}} \subsetneq \mathbb{T}^1_{\mathsf{p}}$. Context-free session types are included in pushdown session types which have only one type identifier X, and where the equation $X\langle\varepsilon\rangle \doteq \mathsf{end}$ accounts for the only occurrence of end. The latter means that the type ends iff the state $X\langle\varepsilon\rangle$ is reached, that is, iff the stack is empty. Thus, we can intuitively think of context-free session types as pushdown types with a single identifier and an empty stack acceptance criterion. This observation points to the fact that the qualifier 'context-free' in the so called context-free session types is a misnomer [9].

The result below summarises the entire hierarchy.

Theorem 1 (Inclusions).

$$\begin{array}{ccccccccccccc}
\mathbb{T}_{\mathsf{f}} & = & \mathbb{T}^0_{\mathsf{p}} & \subsetneq & \mathbb{T}_{\mathsf{r}} & = & \mathbb{T}^0_{\mathsf{n}} & \subsetneq & \mathbb{T}_1 & \subsetneq & \mathbb{T}_{\mathsf{p}} & = & \mathbb{T}_{\mathsf{n}} \subsetneq \mathbb{T}_2 \subsetneq \mathbb{T}_\infty \\
 & & & & & & \rotatebox{90}{$\nsubseteq$}\,\cap & & & & \cup\,\rotatebox{90}{$\subseteq$} & & \\
 & & & & & & \mathbb{T}_{\mathsf{c}} & \subsetneq & \mathbb{T}^1_{\mathsf{p}} & = & \mathbb{T}^1_{\mathsf{n}} \subseteq \mathbb{T}^2_{\mathsf{p}} = \mathbb{T}^2_{\mathsf{n}} \subseteq & \cdots &
\end{array}$$

4 From types to automata

This section describes procedures to convert types in different levels of the hierarchy (recursive systems, 1-counter, pushdown and 2-counter) into automata at the same level. All constructions follow the same guiding principles, so we focus on the bottom level of the hierarchy (recursive systems) and then highlight the main differences as we advance in the hierarchy.

All automata that we consider are *deterministic* and *total*, i.e., the transition functions are such that any input word has a well-defined, unique computation path. We use the alphabet Σ defined in (1). Standard references in automata theory are Hopcroft and Ullman's book [22] and Valiant's PhD thesis [42].

Recursive types and finite-state automata Following the usual notation, a (deterministic) *finite-state automaton* is given by a set Q of states, with a specified initial state $q_0 \in Q$, a transition function $\delta : Q \times \Sigma \to Q$, and a set $A \subseteq Q$ of accepting states. Given a finite word $a_1 a_2 \cdots a_n$, its execution by the automaton yields the sequence of states $s_0, s_1, \ldots, s_n$ where $s_0 = q_0$ and $s_{i+1} = \delta(s_i, a_{i+1})$. A word is *accepted* by the automaton if its execution ends in an accepting state.

The definition of finite-state automata can be augmented into other types of automata. Essentially: in a 1-counter automata we have access to a counter (with operations for incrementing, decrementing, and checking whether the counter is non-zero), in addition to the current state; in a pushdown automata we have access to a stack (with operations for pushing a symbol, popping a symbol, and observing the top symbol of the stack); in a 2-counter automata, we have access to two counters.

Suppose we are given a system of recursive equations $\{X_i \doteq T_i\}_{i \in I}$ over a set $\mathcal{X} = \{X_i\}_{i \in I}$ (which may or may not be contractive, i.e., define a type). Our first step is to convert this system into a normal form in which every right-hand side is either a identifier X, or a single application of one of the type constructors end, $?X.Y$, $!X.Y$, $\&\{\ell\colon X_\ell\}_{\ell \in L}$ or $\oplus\{\ell\colon X_\ell\}_{\ell \in L}$. We can do this by introducing fresh, intermediate identifiers as needed. Essentially, whenever we have an equation $X \doteq ?T_1.T_2$ where T_1, T_2 are not identifiers, we add two new identifiers X', X'', replace the above equation by $X \doteq ?X'.X''$, and add two new equations $X' \doteq T_1$ and $X'' \doteq T_2$. The process is similar for the other type constructors. By doing this repeatedly, we "break down" a long equation into many small equations. The number of new identifiers is linear in the size of the original system of equations.

Given such a system, we construct a finite-state automaton (over the alphabet Σ) as follows. The automaton has a state q_X for every type identifier X, and two additional states: an 'end' state q_{end} and an 'error' state q_{error}. The transitions from q_{error} are described by $q_{\mathsf{error}} \xrightarrow{a} q_{\mathsf{error}}$ for every symbol a. Similarly, the transitions at q_{end} are described by $q_{\mathsf{end}} \xrightarrow{a} q_{\mathsf{error}}$ for every symbol a. The transitions at state q_X are given by the corresponding equation for identifier X, in the obvious way. Some examples:

- If our system contains equation $X \doteq Y$, we have the ε-transition $q_X \xrightarrow{\varepsilon} q_Y$.
- If our system contains $X \doteq !Y.Z$, we have the reading transitions $q_X \xrightarrow{\mathsf{!d}} q_Y$, $q_X \xrightarrow{\mathsf{!c}} q_Z$, and $q_X \xrightarrow{a} q_{\mathsf{error}}$ for any $a \neq \mathsf{!d}, \mathsf{!c}$.
- If our system contains $X \doteq \oplus\{l\colon X, m\colon Y\}$, we have the reading transitions $q_X \xrightarrow{\oplus l} q_X$, $q_X \xrightarrow{\oplus m} q_Y$ and $q_X \xrightarrow{a} q_{\mathsf{error}}$ for any $a \neq \oplus l, \oplus m$.
- If our system contains $X \doteq \mathsf{end}$, we have the reading transitions $q_X \xrightarrow{\mathsf{end}} q_{\mathsf{end}}$ and $q_X \xrightarrow{a} q_{\mathsf{error}}$ for any $a \neq \mathsf{end}$.

We define all states other than q_{error} to be accepting states.[5] Notice that the finite-state automaton described above is an automaton with possible ε-moves. Although, by definition, deterministic finite-state automata do not permit ε-moves, in our case paths of ε-moves are uniquely determined and always reach a state without outgoing ε-transitions (they cannot become stuck in a loop, assuming type contractivity). We can convert the given automaton into an equivalent automaton without ε-moves by 'shortcutting' such moves. Formally, suppose a

[5] We need all states to be accepting, since we might need to look at finite traces to distinguish between two types. For example, $X \doteq \&\{\mathrm{a}\colon X\}$ and $Y \doteq \&\{\mathrm{b}\colon Y\}$ define non-equivalent types that have no finite terminating paths.

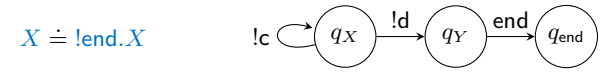


Fig. 8. An automaton for T_{loop} with initial state q_X. All depicted states are accepting.

state X has an outgoing ε-transition to Y; by construction, it is X's only outgoing transition. Assuming X and Y are different states, we can change every transition entering X and make it enter Y instead; finally, we can remove state X (hence removing the ε-transition from X).

We show in Fig. 8 the automaton that corresponds to type T_{loop} (Example 1). Every missing transition points to q_{error} which is not shown. In our examples, all depicted states are accepting, so we omit the usual double circle notation.

1-counter types For 1-counter systems, the only difference in the above construction is that instead of non-parameterised identifiers our equations now involve terms of the form $X\langle \mathsf{z}\rangle$, $X\langle \mathsf{s}\,\mathsf{z}\rangle$, $X\langle N\rangle$, $X\langle \mathsf{s}\,N\rangle$, etc. We assume for simplicity that the identifiers appearing in these equations are restricted as follows: if the left-hand side of an equation is of the form $X\langle \mathsf{z}\rangle$, then the identifiers appearing in the right-hand side must be of the form $X'\langle \mathsf{z}\rangle$ or $X'\langle \mathsf{s}\,\mathsf{z}\rangle$ (with X' possibly different from X); and if the left-hand side of an equation is of the form $X\langle \mathsf{s}\,N\rangle$, then the identifiers appearing in the right-hand side must be of the form $X'\langle N\rangle$, $X'\langle \mathsf{s}\,N\rangle$ or $X'\langle \mathsf{s}\,\mathsf{s}\,N\rangle$. Any system can be converted into this form by adding finitely many new equations, e.g. $X\langle \mathsf{z}\rangle \doteq Y\langle \mathsf{s}\,\mathsf{s}\,\mathsf{s}\,\mathsf{z}\rangle$ can be rewritten as

$$X\langle \mathsf{z}\rangle \doteq X'\langle \mathsf{s}\,\mathsf{z}\rangle \qquad X'\langle \mathsf{s}\,N\rangle \doteq X''\langle \mathsf{s}\,\mathsf{s}\,N\rangle \qquad X''\langle \mathsf{s}\,N\rangle \doteq Y\langle \mathsf{s}\,\mathsf{s}\,N\rangle$$

and $X\langle \mathsf{s}\,N\rangle \doteq Y\langle \mathsf{z}\rangle$ can be rewritten as

$$X\langle \mathsf{s}\,N\rangle \doteq X'\langle N\rangle \qquad X'\langle \mathsf{s}\,N\rangle \doteq X'\langle N\rangle \qquad X'\langle \mathsf{z}\rangle \doteq Y\langle \mathsf{z}\rangle.$$

We can convert a 1-counter type into a (deterministic) 1-counter automaton, so that the transition function depends on whether the counter value is zero (corresponding to a left-hand side of the form $X\langle \mathsf{z}\rangle$) or positive (corresponding to a left-hand side of the form $X\langle \mathsf{s}\,N\rangle$). Furthermore, the changes in the counter value along the identifiers are incorporated by changes in the counter value along the automaton. For example, take equation $X\langle \mathsf{s}\,N\rangle \doteq Y\langle N\rangle$. The corresponding transition from $(q_X, \mathsf{s}, \varepsilon)$ to q_Y decrements the counter.

For illustration purposes, we show how to construct a 1-counter automaton accepting $\mathcal{L}(T_{\text{counter}})$ from Example 2. First, we need to convert the equation for $Y\langle \mathsf{s}\,N\rangle$ into normal form. We add an extra identifier Z and write

$$\begin{array}{ll} X\langle \mathsf{z}\rangle \doteq \&\{\mathsf{inc}\colon X\langle \mathsf{s}\,\mathsf{z}\rangle, \mathsf{dump}\colon Y\langle \mathsf{z}\rangle\} & X\langle \mathsf{s}\,N\rangle \doteq \&\{\mathsf{inc}\colon X\langle \mathsf{s}\,\mathsf{s}\,N\rangle, \mathsf{dump}\colon Y\langle \mathsf{s}\,N\rangle\} \\ Y\langle \mathsf{z}\rangle \doteq \mathsf{end} & Y\langle \mathsf{s}\,N\rangle \doteq !Z\langle \mathsf{s}\,N\rangle.Y\langle N\rangle \\ Z\langle \mathsf{z}\rangle \doteq \mathsf{end} & Z\langle \mathsf{s}\,N\rangle \doteq \mathsf{end} \end{array}$$

The corresponding automaton has states q_X, q_Y, q_Z, one for each type identifier X, Y, Z, as well as an additional state q_{end}. The outgoing transitions for state q_X

$$X\langle \mathsf{z}\rangle \doteq \&\{\mathrm{inc}\colon X\langle \mathsf{s}\,\mathsf{z}\rangle, \mathrm{dump}\colon Y\langle \mathsf{z}\rangle\} \qquad Y\langle \mathsf{z}\rangle \doteq \mathsf{end}$$
$$X\langle \mathsf{s}\,N\rangle \doteq \&\{\mathrm{inc}\colon X\langle \mathsf{s}\,\mathsf{s}\,N\rangle, \mathrm{dump}\colon Y\langle \mathsf{s}\,N\rangle\} \qquad Y\langle \mathsf{s}\,N\rangle \doteq \mathsf{!end}.Y\langle N\rangle$$

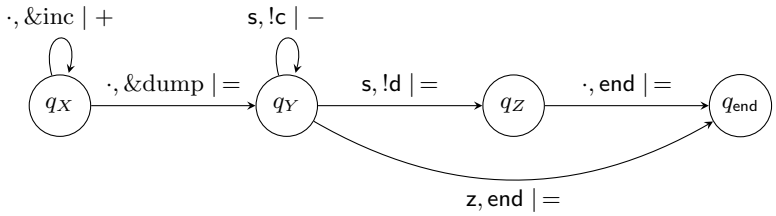


Fig. 9. A 1-counter automaton for type $T_{\text{counter}} = X\langle \mathsf{z}\rangle$. The initial configuration is $(q_X, 0)$. Here a transition $\delta(q, g, a) = (o, q')$ is denoted by an arc from q to q' with label $g, a \mid o$, where $g \in \{\mathsf{z}, \mathsf{s}\}$, $a \in \{\varepsilon\} \cup \Sigma$, and $o \in \{=, +, -\}$. If both $g = \mathsf{z}$ and $g = \mathsf{s}$ lead to the same transition, then we use the symbol · to refer to both transitions. All depicted states are accepting, and non-depicted transitions lead to a non-accepting sink state.

are the same regardless of the counter value: either read &inc, incrementing the counter and staying in q_X; or read &dump, keeping the counter value and moving to q_Y. For state q_Y, if the counter is zero, we can read end while moving to state q_{end}. On the other hand, if the counter is non-zero, we can read !d, keeping the counter value and moving to q_Z; or read !c, decrementing the counter value and staying in q_Y. Finally, for state q_Z we can only read end and move to state q_{end}. Whatever we write in the equation for $Z\langle \mathsf{z}\rangle$ is irrelevant, as this configuration is unreachable. All of this gives the automaton in Fig. 9.

Pushdown types Pushdown systems are similar, but now the behaviour of a identifier is specified by $|\Delta| + 1$ equations, where Δ is the stack alphabet; one equation for each possible symbol at the top of the stack, and one equation for the case that the stack is empty. Accordingly, we use a (deterministic) pushdown automaton to simulate the stack contents by means of push and pop operations. The transitions from a state q_X and a given stack indicator in $\{\varepsilon\} \cup \Delta$ are once more given by the corresponding equation with X as the type identifier on the left-hand side. Fig. 10 shows a pushdown automaton accepting $\mathcal{L}(T_{\text{meta}})$.

2-counter types The translation to 2-counter automata is as for the 1-counter case, but now the behaviour is specified by one of four different cases, depending on which of the two counters is zero or non-zero. Accordingly, we use a (deterministic) 2-counter automaton with the appropriate transition function.

5 From automata to types

The construction in Section 4 explains how we can build an automaton from a system of equations at some level in the hierarchy. If $X\langle \sigma\rangle$ type_{p}, then the

$$X\langle\varepsilon\rangle \doteq \&\{\text{addOut}\colon X\langle\sigma\rangle, \text{addIn}\colon X\langle\tau\rangle\}$$
$$X\langle\sigma S\rangle \doteq \&\{\text{addOut}\colon X\langle\sigma\sigma S\rangle, \text{addIn}\colon X\langle\tau\sigma S\rangle, \text{pop}\colon \mathsf{!end}.X\langle S\rangle\}$$
$$X\langle\tau S\rangle \doteq \&\{\text{addOut}\colon X\langle\sigma\tau S\rangle, \text{addIn}\colon X\langle\tau\tau S\rangle, \text{pop}\colon \mathsf{?end}.X\langle S\rangle\}$$

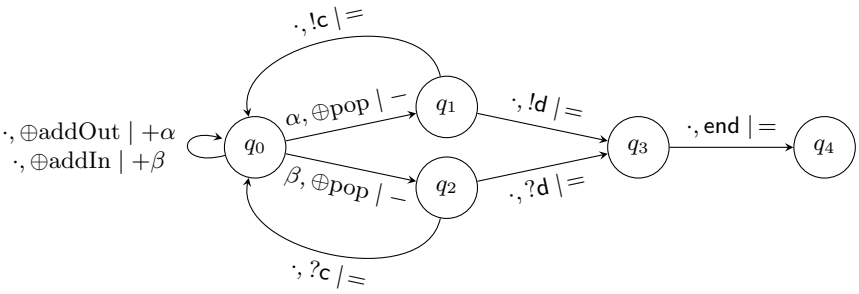


Fig. 10. A pushdown automaton for type $T_{\text{meta}} = X\langle\varepsilon\rangle$. The initial configuration is (q_0, ε). A transition $\delta(q, g, a) = (o, q')$ is denoted by an arc from q to q' with label $g, a \mid o$, where $g \in \{\varepsilon\} \cup \Delta$, $a \in \{\varepsilon\} \cup \Sigma$, and $o \in \text{Op}$. If all choices of g lead to the same transition, we use $\cdot$ to stand for all transitions. All depicted states are accepting.

language of the type given by $X\langle\sigma\rangle$ is the language accepted by the automaton with initial configuration (q_X, σ) (and similarly for recursive, 1-counter, and 2-counter types). Conversely, given an automaton which accepts the language of traces of a type, we can construct the corresponding system of equations that specifies that type. This allow us to obtain a complete correspondence between classes of types and different models of computation based on automata theory. The following result is stronger than previous similar results which only show a forward implication [9]. Recall that a language is said to be *regular* if it is the set of words accepted by some finite-state automaton. We also say that a tree is *regular* if it has a finite number of distinct subtrees.

Theorem 2 (Types, traces and automata).

1. $T\ \text{type}_{\mathsf{r}}$ *iff* $\mathcal{L}(T)$ *is regular iff* $\text{treeof}(T)$ *is regular.*
2. $T\ \text{type}_{1}$ *iff* $\mathcal{L}(T)$ *is accepted by a 1-counter automaton.*
3. $T\ \text{type}_{\mathsf{p}}$ *iff* $\mathcal{L}(T)$ *is a deterministic context-free language.*
4. $T\ \text{type}_{2}$ *iff* $\mathcal{L}(T)$ *is decidable.*

We can now address the decidability of the key problems of type formation, type equivalence and type duality for our various classes of type languages.

Theorem 3 (Decidability results).

1. *Problems* $T\ \text{type}_{\mathsf{r}}$, $T\ \text{type}_{1}$ *and* $T\ \text{type}_{\mathsf{p}}$ *are all decidable in polynomial time.*
2. *Problems* $T \simeq_{\mathsf{r}} U$, $T \simeq_{1} U$ *and* $T \simeq_{\mathsf{p}} U$ *are all decidable.*
3. *Problems* $T \perp_{\mathsf{r}} U$, $T \perp_{1} U$ *and* $T \perp_{\mathsf{p}} U$ *are all decidable.*

We are also able to prove that these problems are undecidable for 2-counter types, since Theorem 2 also provides a construction from automata to systems of equations, and the corresponding problems for automata are undecidable.

Theorem 4 (Undecidability results).
Problems $T\ \mathrm{type}_2$, $T \simeq_2 U$ *and* $T \perp_2 U$ *are all undecidable.*

6 Related work

The first papers on session types by Honda [19] and Takeuchi et al. [38] feature finite types only. Recursive types were introduced later [20] using μ-notation. Gay and Hole [15] introduce algorithms for deciding duality and subtyping of finite-state session types, based on bisimulation. Much of the literature on session types, surveyed by Hüttel et al. [23], uses the same approach. The natural decision algorithms for duality and subtyping presented by Gay and Hole were shown to be exponential in the size of the types by Lange and Yoshida [27], due to reliance on syntactic unfolding. Our polytime complexity for recursive type equivalence follows from the equivalence algorithm for finite-state automata by Hopcroft and Karp [21], and thus has quadratic complexity in the description size, improving on Gay and Hole. Lange and Yoshida use an automata-based algorithm to also achieve quadratic complexity for checking subtyping.

We use a coinductive formulation of infinite session types. This approach has some connections with the work of Keizer et al. [25] who present session types as states of coalgebras. Their types are restricted to finite-state recursive types, but they do address subtyping and non-linear types, two notions that we do not take into consideration. Our coinductive presentation avoids explicitly building coalgebras, and follows Gay et al. [17], solving problems with duality in the presence of recursive types [5,17,28].

We have not addressed the problem of deciding subtyping, but the panorama is not promising. Subtyping is known to be decidable for recursive types $\mathbb{T}_{\mathsf{r}}$ [15] and undecidable for context-free types $\mathbb{T}_{\mathsf{c}}$ [31] or nested types with arity at most one $\mathbb{T}^1_{\mathsf{n}}$ [10], hence for pushdown types with one type constructor $\mathbb{T}^1_{\mathsf{p}}$ (Theorem 1). The undecidability proof of the subtyping problem for context-free session types reduces from the inclusion problem for simple deterministic languages, which was shown to be undecidable by Friedman [13]. That for nested session types reduces from the inclusion problem for Basic Process Algebra [4], which was shown to be undecidable by Groote and Hüttel [18]. Given that 1-counter types $\mathbb{T}_1$ and pushdown types with one type constructor $\mathbb{T}^1_{\mathsf{p}}$ are incomparable (Theorem 1), the problem of subtyping for 1-counter types remains open.

Dependent session types have been studied for binary session types [40,41], for multi-party session types [12,29,45] and for polymorphic, nested session types [9]. Although our parameterised type definitions have some similarities with definitions in some dependently typed systems, we do not support the connection between values in messages and parameters in types, and we have not yet studied how the types that can be expressed in dependent systems fit into our hierarchy.

Connections between multiparty session types and communicating finite-state automata have been explored by Deniélou and Yoshida [11] but the investigation has not been extended to other classes of automata.

Solomon [37] studies the connection between inductive type equality for nested types and language equality for DPDAs and shows that the equivalence problem for nested types is as hard as the equivalence problem for DPDAs, an open problem at the time. We follow a similar approach but define type equivalence as a bisimulation rather than as language equivalence.

Many of the main results in this paper borrow from the theory of automata, developed in the mid-20th century. Here our standard reference is the book by Hopcroft and Ullman [22], where the notions of finite-state, pushdown, and counter automata can be found. 1-counter automata were studied in detail in Valiant's PhD thesis [42]. To prove the equivalence between types and automata, we need to convert automata to satisfying certain properties; similar techniques have appeared in Kao et al. [24] and Valiant and Paterson [43]. Our proofs of decidability of type equivalence make use of the corresponding results for automata [8,21,33,35,36,43]; we specifically mention Sénizergues' impressive result on equivalence of deterministic pushdown automata [36], a work which granted him the Gödel Prize in 2002. Finally, the strict hierarchy results use textbook pumping lemmas for regular languages (due to Rabin and Scott [33]) and context-free languages (due to Bar-Hillel et al. [3] and Kreowski [26]), as well as a somewhat less known result for 1-counter automata (due to Boasson [7]).

7 Conclusion

We introduce different classes of session types, some new, others from the literature, under a uniform framework and place them in n hierarchy. We further study different type-related problems—formation, equivalence and duality—and show that these relations are all decidable up to and including pushdown types.

Much remains to be done. From the point of view of programming languages, one should investigate whether decidability results translate into algorithms that may be incorporated in compilers. Even if subtyping is known to be undecidable for most systems "above" that of recursive types, the problem remains open for 1-counter types, an interesting avenue for further investigation. Our study of classes of infinite types may have applications beyond session types. One promising direction is that of non regular datatypes for functional programming (or polymorphic recursion schemes [30]), such as nested datatypes [6].

We have not addressed the decidability of the type checking problem. Type checking is known to be decidable for finite types, recursive, context-free and nested session types. Given that type checking for nested session types is incorporated in the RAST language [9], a natural first step would be to investigate how to translate 1-counter and pushdown processes into that language.

References

1. Almeida, B., Mordido, A., Thiemann, P., Vasconcelos, V.T.: Polymorphic context-free session types. CoRR **abs/2106.06658** (2021), https://arxiv.org/abs/2106.06658
2. Almeida, B., Mordido, A., Vasconcelos, V.T.: Deciding the bisimilarity of context-free session types. In: TACAS. LNCS, vol. 12079, pp. 39–56. Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_3
3. Bar-Hillel, Y., Perles, M., Shamir, E.: On formal properties of simple phrase structure grammars. Sprachtypologie und Universalienforschung **14**, 143–172 (1961)
4. Bergstra, J.A., Klop, J.W.: Process theory based on bisimulation semantics. In: Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency. LNCS, vol. 354, pp. 50–122. Springer (1988). https://doi.org/10.1007/BFb0013021
5. Bernardi, G., Hennessy, M.: Using higher-order contracts to model session types. Logical Methods in Computer Science **12**(2) (2016). https://doi.org/10.2168/LMCS-12(2:10)2016
6. Bird, R.S., Meertens, L.G.L.T.: Nested datatypes. In: MPC. LNCS, vol. 1422, pp. 52–67. Springer (1998). https://doi.org/10.1007/BFb0054285
7. Boasson, L.: Two iteration theorems for some families of languages. Journal of Computer and System Sciences **7**(6), 583–596 (1973)
8. Böhm, S., Göller, S., Jancar, P.: Equivalence of deterministic one-counter automata is nl-complete. In: STOC. pp. 131–140. ACM (2013). https://doi.org/10.1145/2488608.2488626
9. Das, A., DeYoung, H., Mordido, A., Pfenning, F.: Nested session types. In: ESOP. LNCS, vol. 12648, pp. 178–206. Springer (2021). https://doi.org/10.1007/978-3-030-72019-3_7
10. Das, A., DeYoung, H., Mordido, A., Pfenning, F.: Subtyping on nested polymorphic session types. CoRR **abs/2103.15193** (2021), https://arxiv.org/abs/2103.15193
11. Deniélou, P., Yoshida, N.: Multiparty session types meet communicating automata. In: ESOP. LNCS, vol. 7211, pp. 194–213. Springer (2012). https://doi.org/10.1007/978-3-642-28869-2_10
12. Deniélou, P., Yoshida, N., Bejleri, A., Hu, R.: Parameterised multiparty session types. Log. Methods Comput. Sci. **8**(4) (2012). https://doi.org/10.2168/LMCS-8(4:6)2012
13. Friedman, E.P.: The inclusion problem for simple languages. Theor. Comput. Sci. **1**(4), 297–316 (1976). https://doi.org/10.1016/0304-3975(76)90074-8
14. Gapeyev, V., Levin, M.Y., Pierce, B.C.: Recursive subtyping revealed. J. Funct. Program. **12**(6), 511–548 (2002). https://doi.org/10.1017/S0956796802004318
15. Gay, S.J., Hole, M.: Subtyping for session types in the pi calculus. Acta Inf. **42**(2-3), 191–225 (2005). https://doi.org/10.1007/s00236-005-0177-z
16. Gay, S.J., Poças, D., Vasconcelos, V.T.: The different shades of infinite session types. CoRR **abs/2201.08275** (2022), https://arxiv.org/abs/2201.08275
17. Gay, S.J., Thiemann, P., Vasconcelos, V.T.: Duality of session types: The final cut. In: PLACES. EPTCS, vol. 314, pp. 23–33 (2020). https://doi.org/10.4204/EPTCS.314.3
18. Groote, J.F., Hüttel, H.: Undecidable equivalences for basic process algebra. Inf. Comput. **115**(2), 354–371 (1994). https://doi.org/10.1006/inco.1994.1101
19. Honda, K.: Types for dyadic interaction. In: CONCUR. LNCS, vol. 715, pp. 509–523. Springer (1993). https://doi.org/10.1007/3-540-57208-2_35

20. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: ESOP. LNCS, vol. 1381, pp. 122–138. Springer (1998). https://doi.org/10.1007/BFb0053567
21. Hopcroft, J.E., Karp, R.M.: A linear algorithm for testing equivalence of finite automata. Tech. rep., Cornell University (1971)
22. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley Publishing Company (1979)
23. Hüttel, H., Lanese, I., Vasconcelos, V.T., Caires, L., Carbone, M., Deniélou, P., Mostrous, D., Padovani, L., Ravara, A., Tuosto, E., Vieira, H.T., Zavattaro, G.: Foundations of session types and behavioural contracts. ACM Comput. Surv. **49**(1), 3:1–3:36 (2016). https://doi.org/10.1145/2873052
24. Kao, J.Y., Rampersad, N., Shallit, J.: On NFAs where all states are final, initial, or both. Theoretical Computer Science **410**(47-49), 5010–5021 (2009)
25. Keizer, A.C., Basold, H., Pérez, J.A.: Session coalgebras: A coalgebraic view on session types and communication protocols. In: ESOP. LNCS, vol. 12648, pp. 375–403. Springer (2021). https://doi.org/10.1007/978-3-030-72019-3_14
26. Kreowski, H.J.: A pumping lemma for context-free graph languages. In: International Workshop on Graph Grammars and Their Application to Computer Science. pp. 270–283. Springer (1978)
27. Lange, J., Yoshida, N.: Characteristic formulae for session types. In: TACAS. LNCS, vol. 9636, pp. 833–850. Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_52
28. Lindley, S., Morris, J.G.: Talking bananas: structural recursion for session types. In: ICFP. pp. 434–447. ACM (2016). https://doi.org/10.1145/2951913.2951921
29. de Muijnck-Hughes, J., Brady, E.C., Vanderbauwhede, W.: Value-dependent session design in a dependently typed language. In: PLACES. EPTCS, vol. 291, pp. 47–59 (2019). https://doi.org/10.4204/EPTCS.291.5
30. Mycroft, A.: Polymorphic type schemes and recursive definitions. In: International Symposium on Programming. LNCS, vol. 167, pp. 217–228. Springer (1984). https://doi.org/10.1007/3-540-12925-1_41
31. Padovani, L.: Context-free session type inference. ACM Trans. Program. Lang. Syst. **41**(2), 9:1–9:37 (2019). https://doi.org/10.1145/3229062
32. Pierce, B.C.: Types and programming languages. MIT Press (2002)
33. Rabin, M.O., Scott, D.: Finite automata and their decision problems. IBM journal of research and development **3**(2), 114–125 (1959)
34. Sangiorgi, D.: Introduction to Bisimulation and Coinduction. Cambridge University Press (2012). https://doi.org/10.1017/CBO9780511777110
35. Sénizergues, G.: The equivalence problem for deterministic pushdown automata is decidable. In: ICALP'97. LNCS, vol. 1256, pp. 671–681. Springer (1997). https://doi.org/10.1007/3-540-63165-8_221
36. Sénizergues, G.: L (a)= l(b)? decidability results from complete formal systems. Theoretical Computer Science **251**(1-2), 1–166 (2001)
37. Solomon, M.H.: Type definitions with parameters. In: POPL. pp. 31–38. ACM Press (1978). https://doi.org/10.1145/512760.512765
38. Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing system. In: PARLE. LNCS, vol. 817, pp. 398–413. Springer (1994). https://doi.org/10.1007/3-540-58184-7_118
39. Thiemann, P., Vasconcelos, V.T.: Context-free session types. In: ICFP. pp. 462–475 (2016). https://doi.org/10.1145/2951913.2951926
40. Thiemann, P., Vasconcelos, V.T.: Label-dependent session types. Proc. ACM Program. Lang. **4**(POPL), 67:1–67:29 (2020). https://doi.org/10.1145/3371135

41. Toninho, B., Caires, L., Pfenning, F.: Dependent session types via intuitionistic linear type theory. In: PPDP. pp. 161–172. ACM (2011). https://doi.org/10.1145/2003476.2003499
42. Valiant, L.G.: Decision procedures for families of deterministic pushdown automata. Ph.D. thesis, University of Warwick (1973)
43. Valiant, L.G., Paterson, M.S.: Deterministic one-counter automata. Journal of Computer and System Sciences **10**(3), 340–350 (1975)
44. Vasconcelos, V.T.: Fundamentals of session types. Inf. Comput. **217**, 52–70 (2012). https://doi.org/10.1016/j.ic.2012.05.002
45. Yoshida, N., Deniélou, P., Bejleri, A., Hu, R.: Parameterised multiparty session types. In: FOSSACS. LNCS, vol. 6014, pp. 128–145. Springer (2010). https://doi.org/10.1007/978-3-642-12032-9_10