Characteristic Formulae for Session Types

Julien Lange and Nobuko Yoshida

Imperial College London

Abstract. Subtyping is a crucial ingredient of session type theory and its applications, notably to programming language implementations. In this paper, we study effective ways to check whether a session type is a subtype of another by applying a characteristic formulae approach to the problem. Our core contribution is an algorithm to generate a modal μ-calculus formula that characterises all the supertypes (or subtypes) of a given type. Subtyping checks can then be off-loaded to model checkers, thus incidentally yielding an efficient algorithm to check safety of session types, soundly and completely. We have implemented our theory and compared its cost with other classical subtyping algorithms.

1 Introduction

Motivations Session types [24, 25, 40] have emerged as a fundamental theory to reason about concurrent programs, whereby not only the data aspects of programs are typed, but also their *behaviours* wrt. communication. Recent applications of session types to the reverse-engineering of large and complex distributed systems [12, 29] have led to the need of handling potentially large and complex session types. Analogously to the current trend of modern compilers to rely on external tools such as SMT-solvers to solve complex constraints and offer strong guarantees [16, 23, 31, 32], state-of-the-art model checkers can be used to off-load expensive tasks from session type tools such as [29, 37, 42].

A typical use case for session types in software (reverse-) engineering is to compare the type of an existing program with a candidate replacement, so to ensure that both are "compatible". In this context, a crucial ingredient of session type theory is the notion of *subtyping* [9, 14, 19] which plays a key role to guarantee safety of concurrent programs while allowing for the refinement of specifications and implementations. Subtyping for session types relates to many classical theories such as simulations and pre-orders in automata and process algebra theories; but also to subtyping for recursive types in the λ-calculus [5]. The characteristic formulae approach [1–3, 11, 21, 38, 39], which has been studied since the late eighties as a method to compute simulation-like relations in process algebra and automata, appears then as an evident link between subtyping in session type theory and model checking theories. In this paper, we make the first formal connection between session type and model checking theories, to the best of our knowledge. We introduce a novel approach to session types subtyping based on characteristic formulae; and thus establish that subtyping for session types can be decided in quadratic time wrt. the size of the types.

This improves significantly on the classical algorithm [20]. Subtyping can then be reduced to a model checking problem and thus be discharged to powerful model checkers. Consequently, any advance in model checking technology has an impact on subtyping.

Example Let us illustrate what session types are and what subtyping covers. Consider a simple protocol between a server and a client, from the point of view of the server. The client sends a message of type *request* to the server who decides whether or not the request can be processed by replying *ok* or *ko*, respectively. If the request is rejected, the client is offered another chance to send another request, and so on. This may be described by the *session type* below

$$U_1 = \mathtt{rec}\,\mathbf{x}.\,?\mathit{request}.\{!\mathit{ok}.\mathtt{end} \;\oplus\; !\mathit{ko}.\mathbf{x}\,\} \tag{1}$$

where $\mathtt{rec}\,\mathbf{x}$ binds variable $\mathbf{x}$ in the rest of the type, $?\mathit{msg}$ (resp. $!\mathit{msg}$) specifies the reception (resp. emission) of a message msg, $\oplus$ indicates an *internal choice* between two behaviours, and $\mathtt{end}$ signifies the termination of the conversation. An implementation of a server can then be *type-checked* against U_1.

The client's perspective of the protocol may be specified by the *dual* of U_1:

$$\overline{U}_1 = U_2 = \mathtt{rec}\,\mathbf{x}.\,!\mathit{request}.\{?\mathit{ok}.\mathtt{end} \;\&\; ?\mathit{ko}.\mathbf{x}\,\} \tag{2}$$

where & indicates an *external choice*, i.e., the client expects two possible behaviours from the server. A classical result in session type theory essentially says that if the types of two programs are *dual* of each other, then their parallel composition is free of errors (e.g., deadlock).

Generally, when we say that `integer` is a subtype of `float`, we mean that one can safely use an `integer` when a `float` is required. Similarly, in session type theory, if T is a *subtype* of a type U (written $T \leqslant U$), then T can be used whenever U is required. Intuitively, a type T is a *subtype* of a type U if T is ready to receive no fewer messages than U, and T may not send more messages than U [9, 14]. For instance, we have

$$\begin{array}{l} T_1 = ?\mathit{request}.!\mathit{ok}.\mathtt{end} \;\leqslant\; U_1 \\ T_2 = \mathtt{rec}\,\mathbf{x}.\,!\mathit{request}.\{?\mathit{ok}.\mathtt{end} \,\&\, ?\mathit{ko}.\mathbf{x} \,\&\, ?\mathit{error}.\mathtt{end}\,\} \;\leqslant\; U_2 \end{array} \tag{3}$$

A server of type T_1 can be used whenever a server of type U_1 (1) is required (T_1 is a more refined version of U_1, which always accepts the request). A client of type T_2 can be used whenever a client of type U_2 (2) is required since T_2 is a type that can deal with (strictly) more messages than U_2.

In Section 3.2, we will see that a session type can be naturally transformed into a μ-calculus formula that characterises all its subtypes. The transformation notably relies on the diamond modality to make some branches mandatory, and the box modality to allow some branches to be made optional; see Example 2.

Contribution & synopsis In § 2 we recall session types and give a new abstract presentation of subtyping. In § 3 we present a fragment of the modal μ-calculus and, following [38], we give a simple algorithm to generate a μ-calculus formula from a session type that characterises either all its subtypes or all its supertypes. In § 4, building on results from [9], we give a sound and complete model-checking characterisation of safety for session types. In § 5, we present two other subtyping algorithms for session types: Gay and Hole's classical algorithm [20] based on inference rules that unfold types explicitly; and an adaptation of Kozen et al.'s automata-theoretic algorithm [27]. In § 6, we evaluate the cost of our approach by comparing its performances against the two algorithms from § 5. Our performance analysis is notably based on a tool that generates arbitrary well-formed session types. We conclude and discuss related works in § 7. Due to lack of space, full proofs are relegated to Appendix A (also available online [30]). Our tool and detailed benchmark results are available online [28].

2 Session types and subtyping

Session types are abstractions of the behaviour of a program wrt. the communication of this program on a given *session* (or conversation), through which it interacts with another program (or component).

2.1 Session types

We use a two-party version of the multiparty session types in [15]. For the sake of simplicity, we focus on first order session types (that is, types that carry only simple types (sorts) or values and not other session types). We discuss how to lift this restriction in Section 7. Let $\mathcal{V}$ be a countable set of variables (ranged over by $\mathbf{x}, \mathbf{y}$, etc.); let $\mathbb{A}$ be a (finite) alphabet, ranged over by a, b, etc.; and $\mathcal{A}$ be the set defined as $\{!a \mid a \in \mathbb{A}\} \cup \{?a \mid a \in \mathbb{A}\}$. We let $\dagger$ range over elements of $\{!, ?\}$, so that $\dagger a$ ranges over $\mathcal{A}$. The syntax of session types is given by

$$T \coloneqq \mathtt{end} \;\mid\; \bigoplus_{i \in I} !a_i.\, T_i \;\mid\; \mathop{\&}_{i \in I} ?a_i.\, T_i \;\mid\; \mathtt{rec}\,\mathbf{x}.T \;\mid\; \mathbf{x}$$

where $I \neq \varnothing$ is finite, $a_i \in \mathbb{A}$ for all $i \in I$, $a_i \neq a_j$ for $i \neq j$, and $\mathbf{x} \in \mathcal{V}$. Type $\mathtt{end}$ indicates the end of a session. Type $\bigoplus_{i \in I} !a_i.\, T_i$ specifies an *internal* choice, indicating that the program chooses to send one of the a_i messages, then behaves as T_i. Type $\&_{i \in I} ?a_i.\, T_i$ specifies an *external* choice, saying that the program waits to receive one of the a_i messages, then behaves as T_i. Types $\mathtt{rec}\,\mathbf{x}.T$ and $\mathbf{x}$ are used to specify recursive behaviours. We often write, e.g., $\{!a_1.T_1 \oplus \ldots \oplus !a_k.T_k\}$ for $\bigoplus_{1 \leqslant i \leqslant k} !a_i.T_i$, write $!a_1.T_1$ when $k = 1$, similarly for $\&_{i \in I} ?a_i.\, T_i$, and omit trailing occurrences of $\mathtt{end}$.

The sets of free and bound variables of a type T are defined as usual (the unique binder is the recursion operator $\mathtt{rec}\,\mathbf{x}.T$). For each type T, we assume that two distinct occurrences of a recursion operator bind different variables, and that no variable has both free and bound occurrences. In coinductive definitions, we take an equi-recursive view of types, not distinguishing between a type $\mathtt{rec}\,\mathbf{x}.T$ and its unfolding $T[{}^{\mathtt{rec}\,\mathbf{x}.T}/_{\mathbf{x}}]$. We assume that each type T is *contractive* [34],

$$\frac{j \in I}{\bigoplus_{i\in I} !a_i.\, T_i \xrightarrow{!a_j} T_j}\ \text{[T-OUT]} \qquad \frac{j \in I}{\&_{i\in I} ?a_i.\, T_i \xrightarrow{?a_j} T_j}\ \text{[T-IN]} \qquad \frac{T[{}^{\mathtt{rec}\,\mathtt{x}.T}/_{\mathtt{x}}] \xrightarrow{\dagger a} T'}{\mathtt{rec}\,\mathtt{x}.T \xrightarrow{\dagger a} T'}\ \text{[T-REC]}$$

Fig. 1. LTS for session types in $\mathcal{T}_c$

e.g., $\mathtt{rec}\,\mathtt{x}.\mathtt{x}$ is not a type. Let $\mathcal{T}$ be the set of all (contractive) session types and $\mathcal{T}_c \subseteq \mathcal{T}$ the set of all closed session types (i.e., which do not contain free variables).

A session type $T \in \mathcal{T}_c$ induces a (finite) *labelled transition system* (LTS) according to the rules in Figure 1. We write $T \xrightarrow{\dagger a}$ if there is $T' \in \mathcal{T}$ such that $T \xrightarrow{\dagger a} T'$ and write $T \not\rightarrow$ if $\forall \dagger a \in \mathcal{A} : \neg(T \xrightarrow{\dagger a})$.

2.2 Subtyping for session types

Subtyping for session types was first studied in [19] and further studied in [9,14]. It is a crucial notion for practical applications of session types, as it allows for programs to be *refined* while preserving safety.

We give a definition of subtyping which is parameterised wrt. operators $\oplus$ and $\&$, so to allow us to give a common characteristic formula construction for both the subtype and the supertype relations, cf. Section 3.2. Below, we let $\maltese$ range over $\{\oplus, \&\}$. When writing $\maltese_{i\in I} \dagger a_i.\, T_i$, we take the convention that $\dagger$ refers to ! iff $\maltese$ refers to $\oplus$ (and vice-versa for ? and $\&$). We define the (idempotent) duality operator $\overline{}$ as follows: $\overline{\oplus} \overset{\text{def}}{=} \&$, $\overline{\&} \overset{\text{def}}{=} \oplus$, $\overline{!} \overset{\text{def}}{=} ?$, and $\overline{?} \overset{\text{def}}{=} !$.

Definition 1 (Subtyping). Fix $\maltese \in \{\oplus, \&\}$, $\trianglelefteq^{\maltese} \subseteq \mathcal{T}_c \times \mathcal{T}_c$ is the *largest* relation that contains the rules:

$$\frac{I \subseteq J \quad \forall i \in I : T_i \trianglelefteq^{\maltese} U_i}{\maltese_{i\in I} \dagger a_i.\, T_i \trianglelefteq^{\maltese} \maltese_{j\in J} \dagger a_j.\, U_j}\ \text{[S-}\maltese\text{]} \qquad \frac{}{\mathtt{end} \trianglelefteq^{\maltese} \mathtt{end}}\ \text{[S-END]} \qquad \frac{J \subseteq I \quad \forall j \in J : T_j \trianglelefteq^{\maltese} U_j}{\overline{\maltese}_{i\in I}\, \overline{\dagger} a_i.\, T_i \trianglelefteq^{\maltese} \overline{\maltese}_{j\in J}\, \overline{\dagger} a_j.\, U_j}\ \text{[S-}\overline{\maltese}\text{]}$$

The double line in the rules indicates that the rules should be interpreted *coinductively*. Recall that we are assuming an equi-recursive view of types. $\diamond$

We comment Definition 1 assuming that $\maltese$ is set to $\oplus$. Rule [S-$\maltese$] says that a type $\bigoplus_{j\in J} !a_j.\, U_j$ can be replaced by a type that offers no more messages, e.g., $!a \trianglelefteq^{\oplus} !a \oplus !b$. Rule [S-$\overline{\maltese}$] says that a type $\&_{j\in J} ?a_j.\, U_j$ can be replaced by a type that is ready to receive at least the same messages, e.g., $?a \,\&\, ?b \trianglelefteq^{\oplus} ?a$. Rule [S-END] is trivial. It is easy to see that $\trianglelefteq^{\oplus} = (\trianglelefteq^{\&})^{-1}$. In fact, we can recover the subtyping of [9,14] (resp. [19,20]) from $\trianglelefteq^{\maltese}$, by instantiating $\maltese$ to $\oplus$ (resp. $\&$).

Example 1. Consider the session types from (3), we have $T_1 \trianglelefteq^{\oplus} U_1$, $U_1 \trianglelefteq^{\&} T_1$, $T_2 \trianglelefteq^{\oplus} U_2$, and $U_2 \trianglelefteq^{\&} T_2$.

Hereafter, we will write $\leqslant$ (resp. $\geqslant$) for the pre-order $\trianglelefteq^{\oplus}$ (resp. $\trianglelefteq^{\&}$).

3 Characteristic formulae for subtyping

We give the core construction of this paper: a function that given a (closed) session type T returns a modal μ-calculus formula [26] that characterises either all the supertypes of T or all its subtypes. Technically, we "translate" a session type T into a modal μ-calculus formula ϕ, so that ϕ characterises all the supertypes of T (resp. all its subtypes). Doing so, checking whether T is a subtype (resp. supertype) of U can be reduced to checking whether U is a model of ϕ, i.e., whether $U \models \phi$ holds.

The constructions presented here follow the theory first established in [38]; which gives a characteristic formulae approach for (bi-)simulation-like relations over finite-state processes, notably for CCS processes.

3.1 Modal μ-calculus

In order to encode subtyping for session types as a model checking problem it is enough to consider the fragment of the modal μ calculus below:

$$\phi \coloneqq \top \mid \bot \mid \phi \wedge \phi \mid \phi \vee \phi \mid [\dagger a]\phi \mid \langle \dagger a \rangle \phi \mid \nu \mathbf{x}.\, \phi \mid \mathbf{x}$$

Modal operators $[\dagger a]$ and $\langle \dagger a \rangle$ have precedence over Boolean binary operators $\wedge$ and $\vee$; the greatest fixpoint point operator $\nu\mathbf{x}$ has the lowest precedence (and its scope extends as far to the right as possible). Let $\mathcal{F}$ be the set of all (contractive) modal μ-calculus formulae and $\mathcal{F}_c \subseteq \mathcal{F}$ be the set of all closed formulae. Given a set of actions $A \subseteq \mathcal{A}$, we write $\neg A$ for $\mathcal{A} \backslash A$, and $[A]\phi$ for $\bigwedge_{\dagger a \in A} [\dagger a]\phi$.

The n^{th} approximation of a fixpoint formula is defined as follows:

$$(\nu \mathbf{x}.\, \phi)^0 \stackrel{\text{def}}{=} \top \qquad\qquad (\nu \mathbf{x}.\, \phi)^n \stackrel{\text{def}}{=} \phi[{}^{(\nu \mathbf{x}.\, \phi)^{n-1}}/_{\mathbf{x}}] \qquad \text{if } n > 0$$

A *closed* formula ϕ is interpreted on the labelled transition system induced by a session type T. The satisfaction relation $\models$ between session types and formulae is inductively defined as follows:

$$\begin{array}{lll}
T \models \top & & \\
T \models \phi_1 \wedge \phi_2 & \textit{iff} & T \models \phi_1 \text{ and } T \models \phi_2 \\
T \models \phi_1 \vee \phi_2 & \textit{iff} & T \models \phi_1 \text{ or } T \models \phi_2 \\
T \models [\dagger a]\phi & \textit{iff} & \forall T' \in \mathcal{T}_c \,:\, \text{if } T \xrightarrow{\dagger a} T' \text{ then } T' \models \phi \\
T \models \langle \dagger a \rangle \phi & \textit{iff} & \exists T' \in \mathcal{T}_c \,:\, T \xrightarrow{\dagger a} T' \text{ and } T' \models \phi \\
T \models \nu \mathbf{x}.\, \phi & \textit{iff} & \forall n \geqslant 0 \,:\, T \models (\nu \mathbf{x}.\, \phi)^n
\end{array}$$

Intuitively, $\top$ holds for every T (while $\bot$ never holds). Formula $\phi_1 \wedge \phi_2$ (resp. $\phi_1 \vee \phi_2$) holds if both components (resp. at least one component) of the formula hold in T. The construct $[\dagger a]\phi$ is a *modal* operator that is satisfied if for each $\dagger a$-derivative T' of T, the formula ϕ holds in T'. The dual modality is $\langle \dagger a \rangle \phi$ which holds if there is an $\dagger a$-derivative T' of T such that ϕ holds in T'. Construct $\nu \mathbf{x}.\, \phi$ is the *greatest* fixpoint operator (binding $\mathbf{x}$ in ϕ).

3.2 Characteristic formulae

We now construct a μ-calculus formula from a (closed) session types, parameterised wrt. a constructor $\maltese$. This construction is somewhat reminiscent of the *characteristic functional* of [38].

Definition 2 (Characteristic formulae). The characteristic formulae of $T \in \mathcal{T}_c$ on $\maltese$ is given by function $\boldsymbol{F} : \mathcal{T}_c \times \{\oplus, \&\} \to \mathcal{F}_c$, defined as:

$$\boldsymbol{F}(T, \maltese) \stackrel{\text{def}}{=} \begin{cases} \bigwedge_{i \in I} \langle \dagger\, a_i \rangle \, \boldsymbol{F}(T_i, \maltese) & \text{if } T = \maltese_{i \in I} \, \dagger a_i.\, T_i \\ \bigwedge_{i \in I} [\dagger\, a_i] \, \boldsymbol{F}(T_i, \maltese) & \text{if } T = \overline{\maltese}_{i \in I} \, \dagger a_i.\, T_i \\ \quad \wedge \bigvee_{i \in I} \langle \dagger\, a_i \rangle \top \;\wedge\; [\neg \{\dagger\, a_i \mid i \in I\}] \bot & \\ [\mathcal{A}] \bot & \text{if } T = \mathtt{end} \\ \nu \mathbf{x}.\ \boldsymbol{F}(T', \maltese) & \text{if } T = \mathtt{rec}\, \mathbf{x}. T' \\ \mathbf{x} & \text{if } T = \mathbf{x} \end{cases} \qquad \diamond$$

Given $T \in \mathcal{T}_c$, $\boldsymbol{F}(T, \oplus)$ is a μ-calculus formula that characterises all the *supertypes* of T; while $\boldsymbol{F}(T, \&)$ characterises all its *subtypes*. For the sake of clarity, we comment on Definition 2 assuming that $\maltese$ is set to $\oplus$. The first case of the definition makes every branch *mandatory*. If $T = \bigoplus_{i \in I} !a_i.\, T_i$, then every internal choice branch that T can select must also be offered by a supertype, and the relation must hold after each selection. The second case makes every branch *optional* but requires at least one branch to be implemented. If $T = \&_{i \in I} ?a_i.\, T_i$, then (i) for each of the $?a_i$-branch offered by a supertype, the relation must hold in its $?a_i$-derivative, (ii) a supertype must offer at least one of the $?a_i$ branches, and (iii) a supertype cannot offer anything else but the $?a_i$ branches. If $T = \mathtt{end}$, then a supertype cannot offer any behaviour (recall that $\bot$ does not hold for any type). Recursive types are mapped to greatest fixpoint constructions.

Lemma 1 below states the compositionality of the construction, while Theorem 1, our main result, reduces subtyping checking to a model checking problem. A consequence of Theorem 1 is that the characteristic formula of a session type precisely specifies the set of its subtypes or supertypes.

Lemma 1. $\boldsymbol{F}(T[{}^{U}\!/\mathbf{x}], \maltese) = \boldsymbol{F}(T, \maltese)[{}^{\boldsymbol{F}(U, \maltese)}\!/\mathbf{x}]$

Proof. By structural induction, see Appendix A.1. $\square$

Theorem 1. $\forall T, U \in \mathcal{T}_c : T \trianglelefteq^{\maltese} U \iff U \models \boldsymbol{F}(T, \maltese)$

Proof. The proof essentially follows the techniques of [38], see Appendix A.3. $\square$

Corollary 1. *The following holds:*

(a) $T \leqslant U \iff U \models \boldsymbol{F}(T, \oplus)$
(b) $U \geqslant T \iff T \models \boldsymbol{F}(U, \&)$
(c) $U \models \boldsymbol{F}(T, \oplus) \iff T \models \boldsymbol{F}(U, \&)$

Proof. By Theorem 1 and $\leqslant \,=\, \trianglelefteq^{\oplus}$, $\geqslant \,=\, \trianglelefteq^{\&}$, $\leqslant \,=\, \geqslant^{-1}$, and $\trianglelefteq^{\oplus} = (\trianglelefteq^{\&})^{-1}$ $\square$

Proposition 1. *For all $T, U \in \mathcal{T}_c$, deciding whether or not $U \models \boldsymbol{F}(T, \maltese)$ holds can be done in time complexity of $\mathcal{O}(|T| \times |U|)$, in the worst case; where $|T|$ stands for the number of states in the LTS induced by T.*

Proof. Follows from [11], since the size of $\boldsymbol{F}(T, \maltese)$ increases linearly with $|T|$. ▫

Example 2. Consider session types T_1 and U_1 from (1) and (3) and fix $\mathcal{A} = \{?request, !ok, !ko\}$. Following Definition 2, we obtain:

$$\begin{aligned}\boldsymbol{F}(T_1, \oplus) &= [?request]\langle !ok\rangle[\mathcal{A}]\bot \ \wedge \ \langle ?request\rangle\top \ \wedge \ [\neg\{?request\}]\bot \\ \boldsymbol{F}(U_1, \&) &= \nu\mathbf{x}.\, \langle ?request\rangle\big(([!ok][\mathcal{A}]\bot \ \wedge \ [!ko]\mathbf{x}) \\ &\qquad\qquad \wedge \ (\langle !ok\rangle\top \vee \langle !ko\rangle\top) \ \wedge \ [\neg\{!ok, !ok\}]\bot\big)\end{aligned}$$

We have $U_1 \models \boldsymbol{F}(T_1, \oplus)$ and $T_1 \models \boldsymbol{F}(U_1, \&)$, as expected (recall tat $T_1 \leqslant U_1$).

4 Safety and duality in session types

A key ingredient of session type theory is the notion of *duality* between types. In this section, we study the relation between duality of session types, characteristic formulae, and safety (i.e., error freedom). In particular, building on recent work [9] which studies the preciseness of subtyping for session types, we show how characteristic formulae can be used to guarantee safety. A system (of session types) is a pair of session types T and U that interact with each other by synchronising over messages. We write $T \mid U$ for a system consisting of T and U and let S range over systems of session types.

Definition 3 (Synchronous semantics). The *synchronous* semantics of a *system* of session types $T \mid U$ is given by the rule below, in conjunction with the rules of Figure 1.

$$\frac{T \xrightarrow{\dagger\, a} T' \quad U \xrightarrow{\overline{\dagger} a} U'}{T \mid U \rightarrow T' \mid U'} \ \text{[S-COM]}$$

We write $\rightarrow^*$ for the reflexive transitive closure of $\rightarrow$. ◇

Definition 3 says that two types interact whenever they fire dual operations.

Example 3. Consider the following execution of system $T_1 \mid U_2$, from (3):

$$\begin{aligned}T_1 \mid U_2 \ &= \ ?request.\,!ok.\mathtt{end} \ \mid \ \mathtt{rec}\,\mathbf{x}.\,!request.\{\ldots\} \\ &\longrightarrow \ !ok.\mathtt{end} \ \mid \ \{?ok.\mathtt{end} \ \& \ ?ok.\mathtt{rec}\,\mathbf{x}.\,?request\{\ldots\}\} \ \longrightarrow \ \mathtt{end} \ \mid \ \mathtt{end}\end{aligned}$$

Definition 4 (Error [9] and safety). A system $T_1 \mid T_2$ is an *error* if, either:

(*a*) $T_1 = \maltese_{i\in I} \dagger a_i.\, T_i$ and $T_2 = \maltese_{j\in J} \dagger a_j.\, U_j$, with $\maltese$ fixed;
(*b*) $T_h = \bigoplus_{i\in I} !a_i.\, T_i$ and $T_g = \&_{j\in J} ?a_j.\, U_j$; and $\exists i \in I : \forall j \in J : a_i \neq a_j$, with $h \neq g \in \{1, 2\}$; or
(*c*) $T_h = \mathtt{end}$ and $T_g = \maltese_{i\in I} \dagger a_i.\, T_i$, with $h \neq g \in \{1, 2\}$.

We say that $S = T \mid U$ is *safe* if for all $S' : S \rightarrow^* S'$, S' is not an error. $\diamond$

A system of the form (a) is an error since both types are either attempting to send (resp. receive) messages. An error of type (b) indicates that some of the messages cannot be received by one of the types. An error of type (c) indicates a system where one of the types has terminated while the other still expects to send or receive messages.

Definition 5 (Duality). The dual of a formula $\phi \in \mathcal{F}$, written $\overline{\phi}$ (resp. of a session type $T \in \mathcal{T}$, written $\overline{T}$), is defined recursively as follows:

$$\overline{\phi} \overset{\text{def}}{=} \begin{cases} \overline{\phi_1} \wedge \overline{\phi_2} & \text{if } \phi = \phi_1 \wedge \phi_2 \\ \overline{\phi_1} \vee \overline{\phi_2} & \text{if } \phi = \phi_1 \vee \phi_2 \\ [\overline{\dagger} a]\overline{\phi'} & \text{if } \phi = [\dagger a]\phi' \\ \langle \overline{\dagger} a \rangle \overline{\phi'} & \text{if } \phi = \langle \dagger a \rangle \phi' \\ \nu \mathbf{x}.\, \overline{\phi'} & \text{if } \phi = \nu \mathbf{x}.\, \phi' \\ \phi & \text{if } \phi = \top, \bot, \text{ or } \mathbf{x} \end{cases} \qquad \overline{T} \overset{\text{def}}{=} \begin{cases} \overline{\maltese}_{i \in I}\, \overline{\dagger} a_i.\, \overline{T_i} & \text{if } T = \maltese_{i \in I}\, \dagger a_i.\, T_i \\ \mathtt{rec}\, \mathbf{x}.\overline{T'} & \text{if } T = \mathtt{rec}\, \mathbf{x}.T' \\ \mathbf{x} & \text{if } T = \mathbf{x} \\ \mathtt{end} & \text{if } T = \mathtt{end} \end{cases}$$ $\diamond$

In Definition 5, notice that the dual of a formula only rename labels.

Lemma 2. *For all $T \in \mathcal{T}_c$ and $\phi \in \mathcal{F}_c$, $T \models \phi \iff \overline{T} \models \overline{\phi}$.*

Proof. Direct from the definitions of $\overline{T}$ and $\overline{\phi}$ (labels are renamed uniformly). $\square$

Theorem 2. *For all $T \in \mathcal{T}$: $\overline{\boldsymbol{F}(T, \maltese)} = \boldsymbol{F}(\overline{T}, \overline{\maltese})$.*

Proof. By structural induction on T, see Appendix A.4. $\square$

Theorem 3 follows straightforwardly from [9] and allows us to obtain a sound and complete model-checking based condition for safety, cf. Theorem 4.

Theorem 3 (Safety). *$T \mid U$ is safe $\iff$ $(T \leqslant \overline{U} \vee U \leqslant \overline{T})$.*

Proof. Direction $(\Longrightarrow)$ follows from [9, Table 7] and direction $(\Longleftarrow)$ is by coinduction on the derivations of $T \leqslant \overline{U}$ and $U \leqslant \overline{T}$. See Appendix A.4 for details. $\square$

Finally we achieve:

Theorem 4. *The following statements are equivalent: (a) $T \mid U$ is safe*

(b) $\overline{U} \models \boldsymbol{F}(T, \oplus) \vee \overline{T} \models \boldsymbol{F}(U, \oplus)$

(c) $T \models \boldsymbol{F}(\overline{U}, \&) \vee U \models \boldsymbol{F}(\overline{T}, \&)$

(d) $U \models \boldsymbol{F}(\overline{T}, \&) \vee T \models \boldsymbol{F}(\overline{U}, \&)$

(e) $\overline{T} \models \boldsymbol{F}(U, \oplus) \vee \overline{U} \models \boldsymbol{F}(T, \oplus)$

Proof. By direct applications of Theorem 3, then Corollary 1 and Theorem 2. $\square$

5 Alternative algorithms for subtyping

In order to compare the cost of checking the subtyping relation via characteristic formulae to other approaches, we present two other algorithms: the original algorithm as given by Gay and Hole in [20] and an adaptation of Kozen, Palsberg, and Schwartzbach's algorithm [27] for recursive subtyping for the λ-calculus.

$$\frac{\Gamma, \texttt{rec}\,\mathbf{x}.T \leqslant_c U \vdash T[^{\texttt{rec}\,\mathbf{x}.T}/_{\mathbf{x}}] \leqslant_c U}{\Gamma \vdash \texttt{rec}\,\mathbf{x}.T \leqslant_c U}\ [\text{RL}] \qquad \frac{}{\Gamma \vdash \texttt{end} \leqslant_c \texttt{end}}\ [\text{End}] \qquad \frac{\Gamma, T \leqslant_c \texttt{rec}\,\mathbf{x}.U \vdash T \leqslant_c U[^{\texttt{rec}\,\mathbf{x}.U}/_{\mathbf{x}}]}{\Gamma \vdash T \leqslant_c \texttt{rec}\,\mathbf{x}.U}\ [\text{RR}]$$

$$\frac{I \subseteq J \quad \forall i \in I : \Gamma \vdash T_i \leqslant_c U_i}{\Gamma \vdash \bigoplus_{i \in I} !a_i.\, T_i \leqslant_c \bigoplus_{j \in J} !a_j.\, U_j}\ [\text{Sel}] \qquad \frac{T \leqslant_c U \in \Gamma}{\Gamma \vdash T \leqslant_c U}\ [\text{Assump}] \qquad \frac{J \subseteq I \quad \forall j \in J : \Gamma \vdash T_j \leqslant_c U_j}{\Gamma \vdash \&_{i \in I} ?a_i.\, T_i \leqslant_c \&_{j \in J} ?a_j.\, U_j}\ [\text{Bra}]$$

Fig. 2. Algorithmic subtyping rules [20]

5.1 Gay and Hole's algorithm

The inference rules of Gay and Hole's algorithm are given in Figure 2 (adapted to our setting). The rules essentially follow those of Definition 1 but deal explicitly with recursion. They use judgments $\Gamma \vdash T \leqslant_c U$ in which T and U are (closed) session types and Γ is a sequence of assumed instances of the subtyping relation, i.e., $\Gamma = T_1 \leqslant_c U_1, ..., T_k \leqslant_c U_k$, saying that each pair $T_i \leqslant_c U_i$ has been visited. To guarantee termination, rule [Assump] should always be used if it is applicable.

Theorem 5 (Correspondence [20, Corollary 2]). *$T \leqslant U$ if and only if $\varnothing \vdash T \leqslant_c U$ is derivable from the rules in Figure 2.*

Proposition 2, a contribution of this paper, states the algorithm's complexity.

Proposition 2. *For all $T, U \in \mathcal{T}_c$, the problem of deciding whether or not $\varnothing \vdash T \leqslant_c U$ is derivable has an $\mathcal{O}(n^{2^n})$ time complexity, in the worst case; where n is the number of nodes in the parsing tree of the T or U (which ever is bigger).*

Proof. Assume the bigger session type is T and its size is n (the number of nodes in its parsing tree). Observe that the algorithm in Figure 2 needs to visit every node of T and relies on explicit unfolding of recursive types. Given a type of size n, its unfolding is of size $\mathcal{O}(n^2)$, in the worst case. Hence, we have a chain $\mathcal{O}(n) + \mathcal{O}(n^2) + \mathcal{O}(n^4) + \ldots$, or $\mathcal{O}(\sum_{1 \leqslant i \leqslant k} n^{2^i})$, where k is a bound on the number of derivations needed for the algorithm to terminate. According to [20, Lemma 10], the number of derivations is bounded by the number of sub-terms of T, which is $\mathcal{O}(n)$. Thus, we obtain a worst case time complexity of $\mathcal{O}(n^{2^n})$. ◻

5.2 Kozen, Palsberg, and Schwartzbach's algorithm

Considering that the results of [27] "*generalise to an arbitrary signature of type constructors (. . .)*", we adapt Kozen et al.'s algorithm, originally designed for subtyping recursive types in the λ-calculus. Intuitively, the algorithm reduces the problem of subtyping to checking the language emptiness of an automaton given by the product of two (session) types. The intuition of the theory behind the algorithm is that "*two types are ordered if no common path detects a counterexample*". We give the details of our instantiation below.

The set of type constructors over $\mathcal{A}$, written $\mathfrak{C}_{\mathcal{A}}$, is defined as follows:

$$\mathfrak{C}_{\mathcal{A}} \stackrel{\text{def}}{=} \{\texttt{end}\} \cup \{\oplus_A \mid \varnothing \subset A \subseteq \mathcal{A}\} \cup \{\&_A \mid \varnothing \subset A \subseteq \mathcal{A}\}$$

Definition 6 (Term automata). A term automaton over $\mathcal{A}$ is a tuple $\mathcal{M} = (Q,\, \mathfrak{C}_{\mathcal{A}},\, q_0,\, \delta,\, \ell)$ where

- Q is a (finite) set of states,
- $q_0 \in Q$ is the initial state,
- $\delta : Q \times \mathcal{A} \to Q$ is a (partial) function (the *transition function*), and
- $\ell : Q \to \mathfrak{C}_{\mathcal{A}}$ is a (total) labelling function

such that for any $q \in Q$, if $\ell(q) \in \{\oplus_A, \&_A\}$, then $\delta(q, \dagger a)$ is defined for all $\dagger a \in A$; and for any $q \in Q$ such that $\ell(q) = \texttt{end}$, $\delta(q, \dagger a)$ is undefined for all $\dagger a \in \mathcal{A}$. We decorate Q, δ, etc. with a superscript, e.g., $\mathcal{M}$, where necessary. $\diamond$

We assume that session types have been "translated" to term automata, the transformation is straightforward (see, [15] for a similar transformation). Given a session type $T \in \mathcal{T}_c$, we write $\mathcal{M}(T)$ for its corresponding term automaton.

Definition 7 (Subtyping). $\sqsubseteq$ is the smallest binary relation on $\mathfrak{C}_{\mathcal{A}}$ such that:

$$\texttt{end} \sqsubseteq \texttt{end} \qquad \oplus_A \sqsubseteq \oplus_B \iff A \subseteq B \qquad \&_A \sqsubseteq \&_B \iff B \subseteq A \qquad \diamond$$

Definition 7 essentially maps the rules of Definition 1 to type constructors. The order $\sqsubseteq$ is used in the product automaton to identify final states, see below.

Definition 8 (Product automaton). Given two term automata $\mathcal{M}$ and $\mathcal{N}$ over $\mathcal{A}$, their product automaton $\mathcal{M} \blacktriangleleft \mathcal{N} = (P,\, p_0,\, \Delta,\, F)$ is such that

- $P = Q^{\mathcal{M}} \times Q^{\mathcal{N}}$ are the states of $\mathcal{M} \blacktriangleleft \mathcal{N}$,
- $p_0 = (q_0^{\mathcal{M}}, q_0^{\mathcal{N}})$ is the initial state,
- $\Delta : P \times \mathcal{A} \to P$ is the partial function which for $q_1 \in Q^{\mathcal{M}}$ and $q_2 \in Q^{\mathcal{N}}$ gives
$$\Delta((q_1, q_2), \dagger a) = (\delta^{\mathcal{M}}(q_1, \dagger a), \delta^{\mathcal{N}}(q_2, \dagger a))$$
- $F \subseteq P$ is the set of *accepting* states: $F = \{\, (q_1, q_2) \mid \ell^{\mathcal{M}}(q_1) \not\sqsubseteq \ell^{\mathcal{N}}(q_2) \,\}$

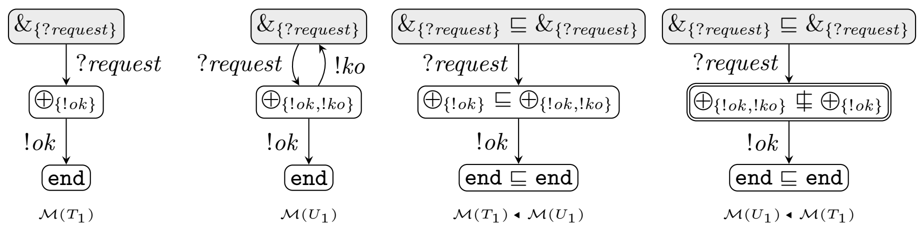
Note that $\Delta((q_1, q_2), \dagger a)$ is defined iff $\delta^{\mathcal{M}}(q_1, \dagger a)$ and $\delta^{\mathcal{N}}(q_2, \dagger a)$ are defined. $\diamond$

Following [27], we obtain Theorem 6.

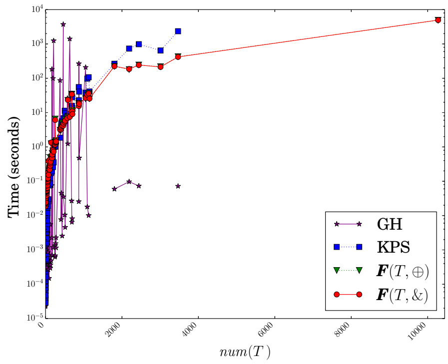
Theorem 6. *Let $T, U \in \mathcal{T}_c$, $T \leqslant U$ iff the language of $\mathcal{M}(T) \blacktriangleleft \mathcal{M}(U)$ is empty.*

Theorem 6 essentially says that $T \leqslant U$ iff one cannot find a "common path" in T and U that leads to nodes whose labels are not related by $\sqsubseteq$, i.e., one cannot find a counterexample for them *not* being in the subtyping relation.

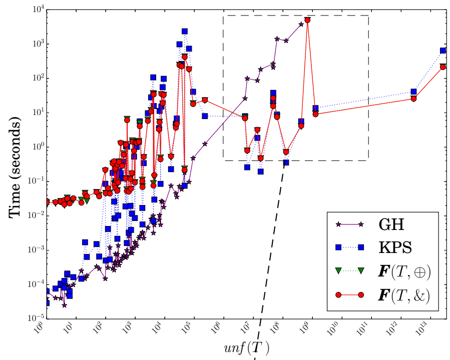
Example 4. Below we show the constructions for T_1 (1) and U_1 (3).



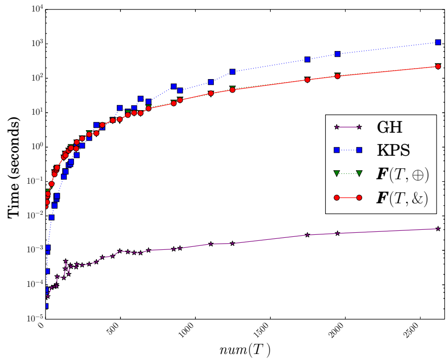
Where initial states are shaded and accepting states are denoted by a double line. Note that the language of $\mathcal{M}(T_1) \blacktriangleleft \mathcal{M}(U_1)$ is empty (no accepting states).



(a) Arbitrary session types (lin. scale)



(b) Arbitrary session types (log. scale)



(c) No recursive definition (lin. scale)

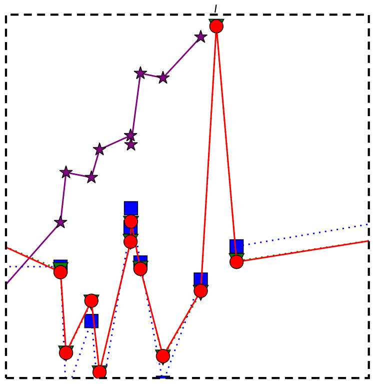


Fig. 3. Benchmarks (1)

Proposition 3. *For all $T, U \in \mathcal{T}_c$, the problem of deciding whether or not the language of $\mathcal{M}(T) \blacktriangleleft \mathcal{M}(U)$ is empty has a worst case complexity of $\mathcal{O}(|T| \times |U|)$; where $|T|$ stands for the number of states in the term automaton $\mathcal{M}(T)$.*

Proof. Follows from the fact that the algorithm in [27] has a complexity of $\mathcal{O}(n^2)$, see [27, Theorem 18]. This complexity result applies also to our instantiation, assuming that checking membership of $\sqsubseteq$ is relatively inexpensive, i.e., $|A| \ll |Q^{\mathcal{M}}|$ for each q such that $\ell^{\mathcal{M}}(q) \in \{\oplus_A, \&_A\}$. $\square$

6 Experimental evaluation

Proposition 2 states that Gay and Hole's classical algorithm has an exponential complexity; while the other approaches have a quadratic complexity (Propositions 1 and 3). The rest of this section presents several experiments that give a better perspective of the *practical* cost of these approaches.

6.1 Implementation overview and metrics

We have implemented three different approaches to checking whether two given session types are in the subtyping relation given in Definition 1. The tool [28], written in Haskell, consists of three main parts: (i) A module that translates session types to the mCRL2 specification language [22] and generates a characteristic (μ-calculus) formula (cf. Definition 2), respectively; (ii) A module implementing the algorithm of [20] (see Section 5.1), which relies on the Haskell `bound` library to make session types unfolding as efficient as possible. (iii) A module implementing our adaptation of Kozen et al.'s algorithm [27], see Section 5.2. Additionally, we have developed an accessory tool which generates arbitrary session types using Haskell's QuickCheck library [10].

The tool invokes the mCRL2 toolset [13] (release version `201409.1`) to check the validity of a μ-calculus formula on a given model. We experimented invoking mCRL2 with several parameters and concluded that the default parameters gave us the best performance overall. Following discussions with mCRL2 developers, we have notably experimented with a parameter that pre-processes the μ-calculus formula to "insert dummy fixpoints in modal operators". This parameter gave us better performances in some cases, but dramatic losses for "super-recursive" session types. Instead, an addition of "dummy fixpoints" while generating the characteristic formulae gave us the best results overall.[1] The tool is thus based on a slight modification of Definition 2 where a modal operator $[\dagger a]\phi$ becomes $[\dagger a]\nu \mathbf{t}.\, \phi$ (with $\mathbf{t}$ fresh and unused) and similarly for $\langle \dagger a \rangle \phi$. Note that this modification does not change the semantics of the generated formulae.

We use the following functions to measure the size of a session type.

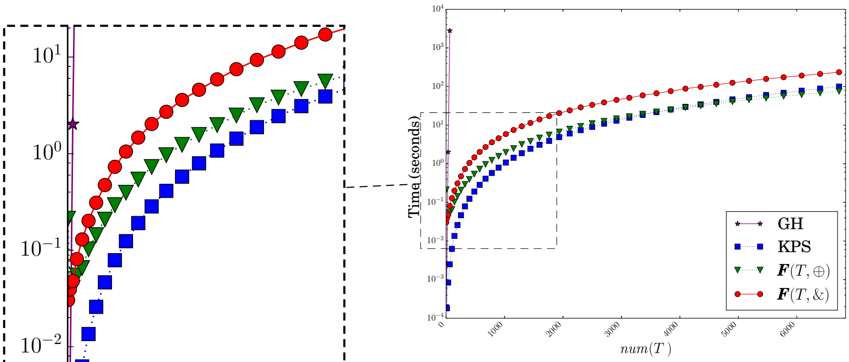
$$
\mathit{num}(T) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } T = \mathtt{end} \text{ or } T = \mathbf{x} \\ \mathit{num}(T') & \text{if } T = \mathtt{rec}\,\mathbf{x}.T' \\ |I| + \sum_{i \in I} \mathit{num}(T_i) & \text{if } T = \maltese_{i \in I} \dagger a_i.\, T_i \end{cases}
\qquad
\mathit{unf}(T) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } T = \mathtt{end} \text{ or } T = \mathbf{x} \\ (1 + |T'|_{\mathbf{x}}) \times \mathit{unf}(T') & \text{if } T = \mathtt{rec}\,\mathbf{x}.T' \\ |I| + \sum_{i \in I} \mathit{unf}(T_i) & \text{if } T = \maltese_{i \in I} \dagger a_i.\, T_i \end{cases}
$$

Function $\mathit{num}(T)$ returns the *number of messages* in T. Letting $|T|_{\mathbf{x}}$ be the number of times variable $\mathbf{x}$ appears *free* in session type T, function $\mathit{unf}(T)$ returns the number of messages in the unfolding of T. Function $\mathit{unf}(T)$ takes into account the structure of a type wrt. recursive definitions and calls (by unfolding once every recursion variable).

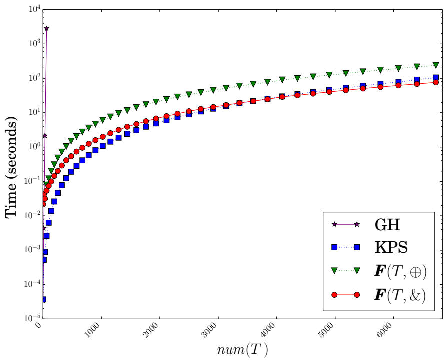
6.2 Benchmark results

The first set of benchmarks compares the performances of the three approaches when the pair of types given are identical, i.e., we measure the time it takes for an algorithm to check whether $T \leqslant T$ holds. The second set of benchmarks considers types that are "unfolded", so that types have different sizes. Note that checking whether two equal types are in the subtyping relation is one of the most costly cases of subtyping since every branch of a choice must be visited.

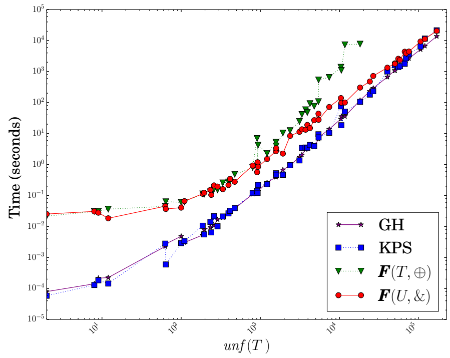
[1] This optimisation was first suggested on the mCRL2 mailing list.



(d) Recursive-Send (lin. scale)



(e) Recursive-Receive (lin. scale)



(f) One-time unfolded (log. scale)

Fig. 4. Benchmarks (2)

Our results below show the performances of four algorithms: (*i*) our Haskell implementation of Gay and Hole's algorithm (GH), (*ii*) our implementation of Kozen, Palsberg, and Schwartzbach's algorithm (KPS), (*iii*) an invocation to mCRL2 to check whether $U \models \boldsymbol{F}(T, \oplus)$ holds, and (*iv*) an invocation to mCRL2 to check whether $T \models \boldsymbol{F}(U, \&)$ holds.

All the benchmarks were conducted on an 3.40GHz Intel i7 computer with 16GB of RAM. Unless specified otherwise, the tests have been executed with a timeout set to 2 hours (7200 seconds). A gap appears in the plots whenever an algorithm reached the timeout. Times (y-axis) are plotted on a *logarithmic* scale, the scale used for the size of types (x-axis) is specified below each plot.

Arbitrary session types Plots (a) and (b) in Figure 3 shows how the algorithms perform with arbitrary session types (randomly generated by our tool). Plot (a) shows clearly that the execution time of KPS, $T \models \boldsymbol{F}(T, \&)$, and $T \models \boldsymbol{F}(T, \oplus)$ mostly depends on $num(T)$; while plot (b) shows that GH is mostly affected by the number of messages in the unfolding of a type ($unf(T)$).

Unsurprisingly, GH performs better for smaller session types, but starts reaching the timeout when $num(T) \approx 700$. The other three algorithms have roughly similar performances, with the model checking based ones performing slightly better for large session types. Note that both $T \models \boldsymbol{F}(T, \&)$ and $T \models \boldsymbol{F}(T, \oplus)$ have roughly the same execution time.

Non-recursive arbitrary session types Plot (c) in Figure 3 shows how the algorithms perform with arbitrary types that do *not* feature any recursive definition (randomly generated by our tool), i.e., the types are of the form:

$$T \coloneqq \mathtt{end} \;\mid\; \bigoplus_{i \in I} \,!a_i.\, T_i \;\mid\; \&_{i \in I} \,?a_i.\, T_i$$

The plot shows that GH performs much better than the other three algorithms (terminating under 1s for each invocation). Indeed this set of benchmarks is the best case scenario for GH: there is no recursion hence no need to unfold types. Observe that the model checking based algorithms perform better than KPS for large session types. Again, $T \models \boldsymbol{F}(T, \&)$ and $T \models \boldsymbol{F}(T, \oplus)$ behave similarly.

Handcrafted session types Plots (d) and (e) in Figure 4 shows how the algorithms deal with "super-recursive" types, i.e., types of the form:

$$T \coloneqq \mathtt{rec}\, \mathbf{x}_1.\dagger a_1. \ldots \mathtt{rec}\, \mathbf{x}_k.\dagger a_k \left\{ \maltese_{1 \leqslant i \leqslant k} \dagger a_i.\{\maltese_{1 \leqslant j \leqslant k} \dagger a_j.\mathbf{x}_j\} \right\}$$

where $num(T) = k(k+2)$ for each T. Plot (d) shows the results of experiments with $\maltese$ set to $\oplus$ and $\dagger$ to $!$; while $\maltese$ is set to $\&$ and $\dagger$ to $?$ in plot (e).

The exponential time complexity of GH appears clearly in both plots: GH starts reaching the timeout when $num(T) = 80$ ($k = 8$). However, the other three algorithms deal well with larger session types of this form. Interestingly, due to the nature of these session types (consisting of either only *internal* choices or only *external* choices), the two model checking based algorithms perform slightly differently. This is explained by Definition 2 where the formula generated with $\boldsymbol{F}(T, \&)$ for an internal choice is larger than for an external choice, and vice-versa for $\boldsymbol{F}(T, \oplus)$. Observe that, $T \models \boldsymbol{F}(T, \oplus)$ (resp. $T \models \boldsymbol{F}(T, \&)$) performs better than KPS for large session types in plot (d) (resp. plot (e)).

Unfolded types The last set of benchmarks evaluates the performances of the four algorithms to check whether $T = \mathtt{rec}\, \mathbf{x}.V \leqslant \mathtt{rec}\, \mathbf{x}.\, (V[{}^{V}\!/\mathbf{x}]) = U$ holds, where $\mathbf{x}$ is fixed and V (randomly generated) is of the form:

$$V \coloneqq \bigoplus_{i \in I} \,!a_i.\, V_i \;\mid\; \&_{i \in I} \,?a_i.\, V_i \;\mid\; \mathbf{x}$$

Plots (f) in Figure 4 shows the results of our experiments (we have set the timeout to 6 hours for these tests). Observe that $U \models \boldsymbol{F}(T, \oplus)$ starts reaching the timeout quickly. In this case, the model (i.e., U) is generally much larger than the formula (i.e., $\boldsymbol{F}(T, \oplus)$). After discussing with the mCRL2 team, this discrepancy seems to originate from internal optimisations of the model checker that can be diminished (or exacerbated) by tweaking the parameters of the toolset. The other three algorithms have similar performances. Note that the good performance of GH in this case can be explained by the fact that there is only one recursion variable in these types; hence the size of their unfolding does not grow very fast.

7 Related work and conclusions

Related work Subtyping for recursive types has been studied for many years. Amadio and Cardelli [5] introduced the first subtyping algorithm for recursive types for the λ-calculus. Kozen et al. gave a quadratic subtyping algorithm in [27], which we have adapted for session types, cf. Section 5.2. A good introduction to the theory and history of the field is in [18]. Pierce and Sangiori [35] introduced subtyping for IO types in the π-calculus, which later became a foundation for the algorithm of Gay and Hole who first introduced subtyping for session types in the π-calculus in [20]. The paper [14] studied an abstract encoding between linear types and session types, with a focus on subtyping. Chen et al. [9] studied the notion of *preciseness* of subtyping relations for session types. The present work is the first to study the algorithmic aspect of the problem.

Characteristic formulae for finite processes were first studied in [21], then in [38] for finite-state processes. Since then the theory has been studied extensively [1–3, 11, 17, 33, 39] for most of the van Glabbeek's spectrum [41] and in different settings (e.g., time [4] and probabilistic [36]). See [2, 3] for a detailed historical account of the field. This is the first time characteristic formulae are applied to the field of session types. A recent work [3] proposes a general framework to obtain characteristic formula constructions for simulation-like relation "for free". We chose to follow [38] as it was a better fit for session types as they allow for a straightforward inductive construction of a characteristic formula. Moreover, [38] uses the standard μ-calculus which allowed us to integrate our theory with an existing model checker.

Conclusions In this paper, we gave a first connection between session types and model checking, through a characteristic formulae approach based on the μ-calculus. We gave three new algorithms for subtyping: two are based on model checking and one is an instantiation of an algorithm for the λ-calculus [27]. All of which have a quadratic complexity in the worst case and behave well in practice.

Our approach can be easily: (*i*) adapted to types for the λ-calculus (see Appendix B) and (*ii*) extended to session types that carry other (*closed*) session types, e.g., see [9, 20], by simply applying the algorithm recursively on the carried types. For instance, to check $!a\langle ?c \,\&\, ?d\rangle \leqslant\, !a\langle ?c\rangle \oplus\, !b\langle \mathtt{end}\rangle$ one can check the subtyping for the outer-most types, while building constraints, i.e., $\{?c \,\&\, ?d \leqslant ?c\}$, to be checked later on, by re-applying the algorithm.

The present work paves the way for new connections between session types and modal fixpoint logic or model checking theories. It is a basis for upcoming connections between model checking and classical problems of session types, such as the asynchronous subtyping of [9] and multiparty compatibility checking [15, 29]. We are also considering applying model checking approaches to session types with probabilistic, logical [6], or time [7,8] annotations. Finally, we remark that [9] also establishes that subtyping (cf. Definition 1) is *sound* (but not complete) wrt. the *asynchronous* semantics of session types, which models programs that communicate through FIFO buffers. Thus, our new conditions (items (*b*)-(*e*) of Theorem 4) also imply safety (*a*) in the asynchronous setting.

Acknowledgements We would like to thank Luca Aceto, Laura Bocchi, and Alceste Scalas for their invaluable comments on earlier versions of this work. This work is partially supported by UK EPSRC projects EP/K034413/1, EP/K011715/1, and EP/L00058X/1; and by EU 7FP project under grant agreement 612985 (UPSCALE).

References

1. L. Aceto and A. Ingólfsdóttir. A characterization of finitary bisimulation. *Inf. Process. Lett.*, 64(3):127–134, 1997.
2. L. Aceto and A. Ingólfsdóttir. Characteristic formulae: From automata to logic. *Bulletin of the EATCS*, 91:58–75, 2007.
3. L. Aceto, A. Ingólfsdóttir, P. B. Levy, and J. Sack. Characteristic formulae for fixed-point semantics: a general framework. *Mathematical Structures in Computer Science*, 22(2):125–173, 2012.
4. L. Aceto, A. Ingólfsdóttir, M. L. Pedersen, and J. Poulsen. Characteristic formulae for timed automata. *ITA*, 34(6):565–584, 2000.
5. R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Trans. Program. Lang. Syst.*, 15(4):575–631, 1993.
6. L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR 2010*, pages 162–176, 2010.
7. L. Bocchi, J. Lange, and N. Yoshida. Meeting deadlines together. In *CONCUR 2015*, pages 283–296, 2015.
8. L. Bocchi, W. Yang, and N. Yoshida. Timed multiparty session types. In *CONCUR 2014*, pages 419–434, 2014.
9. T.-C. Chen, M. Dezani-Ciancaglini, and N. Yoshida. On the preciseness of subtyping in session types. In *PPDP 2014*, pages 146–135. ACM Press, 2014.
10. K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of Haskell programs. In *ICFP 2000*, pages 268–279, 2000.
11. R. Cleaveland and B. Steffen. Computing behavioural relations, logically. In *ICALP 1991*, pages 127–138, 1991.
12. Cognizant. Zero Deviation Lifecycle. `http://www.zdlc.co`.
13. S. Cranen, J. F. Groote, J. J. A. Keiren, F. P. M. Stappers, E. P. de Vink, W. Wesselink, and T. A. C. Willemse. An overview of the mCRL2 toolset and its recent advances. In *TACAS 2013*, pages 199–213, 2013.
14. R. Demangeon and K. Honda. Full abstraction in a subtyped pi-calculus with linear types. In *CONCUR 2011*, pages 280–296, 2011.
15. P. Deniélou and N. Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In *ICALP 2013*, pages 174–186, 2013.
16. I. S. Diatchki. Improving Haskell types with SMT. In *Haskell 2015*, pages 1–10. ACM, 2015.
17. H. Fecher and M. Steffen. Characteristic mu-calculus formulas for underspecified transition systems. *Electr. Notes Theor. Comput. Sci.*, 128(2):103–116, 2005.
18. V. Gapeyev, M. Y. Levin, and B. C. Pierce. Recursive subtyping revealed. *J. Funct. Program.*, 12(6):511–548, 2002.
19. S. J. Gay and M. Hole. Types and subtypes for client-server interactions. In *ESOP 2009*, pages 74–90, 1999.
20. S. J. Gay and M. Hole. Subtyping for session types in the pi calculus. *Acta Inf.*, 42(2-3):191–225, 2005.
21. S. Graf and J. Sifakis. A modal characterization of observational congruence on finite terms of CCS. *Information and Control*, 68(1-3):125–145, 1986.
22. J. F. Groote and M. R. Mousavi. *Modeling and analysis of communicating systems.* MIT Press, 2014.
23. A. Gundry. A typechecker plugin for units of measure: Domain-specific constraint solving in GHC Haskell. In *Haskell 2015*, pages 11–22. ACM, 2015.

24. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP 1998*, pages 122–138, 1998.
25. H. Hüttel, I. Lanese, V. T. Vasconcelos, L. Caires, M. Carbone, P.-M. Deniélou, D. Mostrous, L. Padovani, A. Ravara, E. Tuosto, et al. Foundations of behavioural types. *Report of the EU COST Action IC1201 (BETTY)*, 2014. `www.behavioural-types.eu/publications/WG1-State-of-the-Art.pdf`.
26. D. Kozen. Results on the propositional mu-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.
27. D. Kozen, J. Palsberg, and M. I. Schwartzbach. Efficient recursive subtyping. *Mathematical Structures in Computer Science*, 5(1):113–125, 1995.
28. J. Lange. Tool and benchmark data. `http://bitbucket.org/julien-lange/modelcheckingsessiontypesubtyping`, 2015.
29. J. Lange, E. Tuosto, and N. Yoshida. From communicating machines to graphical choreographies. In *POPL 2015*, pages 221–232, 2015.
30. J. Lange and N. Yoshida. Full version of this paper. `www.doc.ic.ac.uk/~jlange/papers/char-formula-subtyping.pdf`, 2015.
31. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR-16 2010*, pages 348–370, 2010.
32. K. R. M. Leino and K. Yessenov. Stepwise refinement of heap-manipulating code in Chalice. *Formal Asp. Comput.*, 24(4-6):519–535, 2012.
33. M. Müller-Olm. Derivation of characteristic formulae. *Electr. Notes Theor. Comput. Sci.*, 18:159–170, 1998.
34. B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.
35. B. C. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–453, 1996.
36. J. Sack and L. Zhang. A general framework for probabilistic characterizing formulae. In *VMCAI 2012*, pages 396–411, 2012.
37. Scribble Project homepage. `www.scribble.org`.
38. B. Steffen. Characteristic formulae. In *ICALP 1989*, pages 723–732, 1989.
39. B. Steffen and A. Ingólfsdóttir. Characteristic formulae for processes with divergence. *Inf. Comput.*, 110(1):149–163, 1994.
40. K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *PARLE 1994*, pages 398–413, 1994.
41. R. J. van Glabbeek. The linear time-branching time spectrum (extended abstract). In *CONCUR 1990*, pages 278–297, 1990.
42. N. Yoshida, R. Hu, R. Neykova, and N. Ng. The Scribble protocol language. In *TGC 2013*, volume 8358, pages 22–41. Springer, 2013.

A Appendix: Proofs

A.1 Compositionality

Lemma 1. $\boldsymbol{F}(T[{}^{U}/_{\mathbf{x}}], \maltese) = \boldsymbol{F}(T, \maltese)[{}^{\boldsymbol{F}(U,\maltese)}/_{\mathbf{x}}]$

Proof. By structural induction on the structure of T.

1. If $T = \mathtt{end}$, then
   - $T[{}^{U}/_{\mathbf{x}}] = \mathtt{end}$ and $\boldsymbol{F}(T[{}^{U}/_{\mathbf{x}}], \maltese) = [\mathcal{A}]\bot$, and
   - $\boldsymbol{F}(T, \maltese) = [\mathcal{A}]\bot = [\mathcal{A}]\bot[{}^{\boldsymbol{F}(U,\maltese)}/_{\mathbf{x}}]$.
2. If $T = \mathbf{x}$, then
   - $T[{}^{U}/_{\mathbf{x}}] = U$, hence $\boldsymbol{F}(T[{}^{U}/_{\mathbf{x}}], \maltese) = \boldsymbol{F}(U, \maltese)$, and
   - $\boldsymbol{F}(T, \maltese) = \mathbf{x}$, hence $\boldsymbol{F}(T, \maltese)[{}^{\boldsymbol{F}(U,\maltese)}/_{\mathbf{x}}] = \boldsymbol{F}(U, \maltese)$.
3. If $T = \mathbf{y}(\neq \mathbf{x})$, then
   - $\boldsymbol{F}(\mathbf{y}[{}^{U}/_{\mathbf{x}}], \maltese) = \boldsymbol{F}(\mathbf{y}, \maltese) = \mathbf{y}$, and
   - $\boldsymbol{F}(\mathbf{y}, \maltese)[{}^{\boldsymbol{F}(U,\maltese)}/_{\mathbf{x}}] = \mathbf{y}[{}^{\boldsymbol{F}(U,\maltese)}/_{\mathbf{x}}] = \mathbf{y}$.
4. If $T = \maltese_{i\in I}\, \dagger a_i . T_i$, then

$$\begin{aligned}
\boldsymbol{F}(T[{}^{U}/_{\mathbf{x}}], \maltese) &= \boldsymbol{F}(\maltese_{i\in I}\ \dagger a_i . T_i[{}^{U}/_{\mathbf{x}}], \maltese) \\
&= \bigwedge_{i\in I} \langle \dagger\, a_i \rangle\, \boldsymbol{F}(T_i[{}^{U}/_{\mathbf{x}}], \maltese) \\
\text{\tiny (I.H.)} \quad &= \bigwedge_{i\in I} \langle \dagger\, a_i \rangle \left(\boldsymbol{F}(T_i, \maltese)[{}^{\boldsymbol{F}(U,\maltese)}/_{\mathbf{x}}] \right) \\
&= \left(\bigwedge_{i\in I} \langle \dagger\, a_i \rangle\, \boldsymbol{F}(T_i, \maltese) \right) [{}^{\boldsymbol{F}(U,\maltese)}/_{\mathbf{x}}] \\
&= \boldsymbol{F}(T, \maltese)[{}^{\boldsymbol{F}(U,\maltese)}/_{\mathbf{x}}]
\end{aligned}$$

5. If $T = \overline{\maltese}_{i\in I}\, \dagger a_i . T_i$, then

$$\begin{aligned}
\boldsymbol{F}(T[{}^{U}/_{\mathbf{x}}], \maltese) &= \boldsymbol{F}(\overline{\maltese}_{i\in I}\ \dagger a_i . T_i[{}^{U}/_{\mathbf{x}}], \maltese) \\
&= \bigwedge_{i\in I} [\dagger\, a_i]\, \boldsymbol{F}(T_i[{}^{U}/_{\mathbf{x}}], \maltese) \;\wedge \\
&\qquad \bigvee_{i\in I} \langle \dagger\, a_i \rangle \top \;\wedge\; [\neg\{\dagger\, a_i \mid i \in I\}]\bot \\
\text{\tiny (I.H.)} \quad &= \bigwedge_{i\in I} [\dagger\, a_i] \left(\boldsymbol{F}(T_i, \maltese)[{}^{\boldsymbol{F}(U,\maltese)}/_{\mathbf{x}}] \right) \;\wedge \\
&\qquad \bigvee_{i\in I} \langle \dagger\, a_i \rangle \top \;\wedge\; [\neg\{\dagger\, a_i \mid i \in I\}]\bot \\
&= \left(\bigwedge_{i\in I} [\dagger\, a_i]\, \boldsymbol{F}(T_i, \maltese) \;\wedge \right. \\
&\qquad \left. \bigvee_{i\in I} \langle \dagger\, a_i \rangle \top \;\wedge\; [\neg\{\dagger\, a_i \mid i \in I\}]\bot \right) [{}^{\boldsymbol{F}(U,\maltese)}/_{\mathbf{x}}] \\
&= \boldsymbol{F}(T, \maltese)[{}^{\boldsymbol{F}(U,\maltese)}/_{\mathbf{x}}]
\end{aligned}$$

6. If $T = \mathtt{rec}\,\mathbf{y}.T'$ $(\mathbf{x} \neq \mathbf{y})$, we have

$$\begin{aligned} \boldsymbol{F}(\mathtt{rec}\,\mathbf{y}.T'[{}^{U}\!/\mathbf{x}], \maltese) &= \nu\mathbf{y}.\ \boldsymbol{F}(T'[{}^{U}\!/\mathbf{x}], \maltese) \\ {\scriptstyle (I.H.)} \quad &= \nu\mathbf{y}.\ \boldsymbol{F}(T', \maltese)[{}^{\boldsymbol{F}(U,\maltese)}\!/\mathbf{x}] \\ &= \boldsymbol{F}(T, \maltese)[{}^{\boldsymbol{F}(U,\maltese)}\!/\mathbf{x}] \end{aligned}$$

□

A.2 Extensions and approximations

The proofs in this section follow closely the proof techniques in [38].

Definition 9 (Extended subtyping). Let $T, U \in \mathcal{T}$, $\phi \in \mathcal{F}$, and $\vec{\mathbf{x}} = (\mathbf{x}_1, \dots, \mathbf{x}_n)$ be a vector containing all the free variables in T, U, or ϕ. We define the *extended subtyping* $\leqslant_e$ and the *extended satisfaction relation*, $\models_e$, by

1. $T \leqslant_e U \iff \forall \vec{V} \in \mathcal{T}^n : T[{}^{\vec{V}}\!/\vec{\mathbf{x}}] \leqslant U[{}^{\vec{V}}\!/\vec{\mathbf{x}}]$
2. $T \models_e \phi \iff \forall \vec{V} \in \mathcal{T}^n \forall \vec{\psi} \in \mathcal{F}^n : \vec{V} \models \vec{\psi} \implies T[{}^{\vec{V}}\!/\vec{\mathbf{x}}] \models \phi[{}^{\vec{\psi}}\!/\vec{\mathbf{x}}]$

where $\vec{V} \models \vec{\psi}$ is understood component wise. ◇

Definition 10 (Subtyping approximations). Let $T, U \in \mathcal{T}$ and $\vec{\mathbf{x}} = (\mathbf{x}_1, \dots, \mathbf{x}_n)$ be a vector containing all the free variables in T or U. The extended k-limited subtyping, $\leqslant_{e,k}$ is defined inductively on k as follows: $T \leqslant_{e,0} U$ always holds; if $k \geqslant 1$, then $T \leqslant_{e,k} U$ holds iff for all $\vec{V} \in \mathcal{T}^n$, $T[{}^{\vec{V}}\!/\vec{\mathbf{x}}] \leqslant_{e,k} U[{}^{\vec{V}}\!/\vec{\mathbf{x}}]$ can be derived from the following rules:

$$\frac{I \subseteq J \quad \forall i \in I : T_i \trianglelefteq^{\maltese}_{e,k-1} U_i}{\maltese_{i \in I} \dagger a_i . T_i \trianglelefteq^{\maltese}_{e,k} \maltese_{j \in J} \dagger a_j . U_j} \text{ [S-OUT]} \qquad \frac{J \subseteq I \quad \forall j \in J : T_j \trianglelefteq^{\maltese}_{e,k-1} U_j}{\overline{\maltese}_{i \in I} \dagger a_i . T_i \trianglelefteq^{\maltese}_{e,k} \overline{\maltese}_{j \in J} \dagger a_j . U_j} \text{ [S-IN]}$$

$$\frac{}{\mathsf{end} \trianglelefteq^{\maltese}_{e,k} \mathsf{end}} \text{ [S-END]}$$

Recall that we are assuming an equi-recursive view of types. ◇

Lemma 3. $T \trianglelefteq^{\maltese}_{e} U \iff \forall k : T \trianglelefteq^{\maltese}_{e,k} U$

Proof. The ($\Rightarrow$) direction is straightforward, while the converse follow from the fact the session types we consider have only a finite number of states. □

Definition 11 (Semantics approximations). Let $T \in \mathcal{T}$ and $\vec{\mathbf{x}} = (\mathbf{x}_1, \dots, \mathbf{x}_n)$ be a vector containing all the free variables in T. The extended k-limited satisfaction relation $\models_{e,k}$ is defined inductively as follows on k: $T \models_{e,0} \phi$ always holds; if $k \geqslant 1$, then $\models_{e,k}$ is given by:

$$\begin{array}{lll} T \models_{e,k} \top & & \\ T \models_{e,k} \phi_1 \wedge \phi_2 & \textit{iff} & T \models_{e,k} \phi_1 \text{ and } T \models_{e,k} \phi_2 \\ T \models_{e,k} \phi_1 \vee \phi_2 & \textit{iff} & T \models_{e,k} \phi_1 \text{ or } T \models_{e,k} \phi_2 \\ T \models_{e,k} [\dagger a]\phi & \textit{iff} & \forall \vec{V} \in \mathcal{T}^n\ \forall T' : \text{if } T[{}^{\vec{V}}\!/\vec{\mathbf{x}}] \xrightarrow{\dagger a} T' \text{ then } T' \models_{e,k-1} \phi \\ T \models_{e,k} \langle \dagger a \rangle \phi & \textit{iff} & \forall \vec{V} \in \mathcal{T}^n\ \exists T' : T[{}^{\vec{V}}\!/\vec{\mathbf{x}}] \xrightarrow{\dagger a} T' \text{ and } T' \models_{e,k-1} \phi \\ T \models_{e,k} \nu\mathbf{x}.\, \phi & \textit{iff} & \forall n : T \models_{e,k} (\nu\mathbf{x}.\, \phi)^n \end{array}$$

◇

Lemma 4. $T \models_e \phi \iff \forall k \geqslant 0 : T \models_{e,k} \phi$

Proof. The ($\Rightarrow$) direction is straightforward, while the ($\Leftarrow$) direction follows from the fact that a session type induce a finite LTS. $\square$

Lemma 5 (Fixpoint properties). *Let $T \in \mathcal{T}$ and $\phi \in \mathcal{F}$, then we have:*

1. $T \models_{e,k} \nu\mathbf{x}.\,\phi \iff T \models_{e,k} \phi[{}^{\nu\mathbf{x}.\,\phi}/_{\mathbf{x}}]$
2. $\mathtt{rec}\,\mathbf{x}.T \models_{e,k} \phi \iff T[{}^{\mathtt{rec}\,\mathbf{x}.T}/_{\mathbf{x}}] \models_{e,k} \phi$
3. $\mathtt{rec}\,\mathbf{x}.T \trianglelefteq^{\maltese}_{e,k} T[{}^{\mathtt{rec}\,\mathbf{x}.T}/_{\mathbf{x}}] \trianglelefteq^{\maltese}_{e,k} \mathtt{rec}\,\mathbf{x}.T$

Proof. The first property is a direct consequence of the definition of $\models_{e,k}$, while the last two properties follow from the equi-recursive view of types. $\square$

A.3 Main results

Theorem 1. $\forall T, U \in \mathcal{T}_c : T \trianglelefteq^{\maltese} U \iff U \models \boldsymbol{F}(T, \maltese)$

Proof. Direct consequence of Lemma 6. $\square$

Lemma 6 (Main lemma). $\forall T, U \in \mathcal{T} : T \trianglelefteq^{\maltese}_e U \iff U \models_e \boldsymbol{F}(T, \maltese)$

Proof. According to Lemmas 3 and 4, it is enough to show that

$$\forall k \geqslant 0 : \forall U, T \in \mathcal{T} : T \trianglelefteq^{\maltese}_{e,k} U \iff U \models_{e,k} \boldsymbol{F}(T, \maltese) \tag{4}$$

We show this by induction on k. If $k = 0$, the result holds trivially, let us show that it also holds for $k \geqslant 1$. We distinguish four cases according to the structure of T.

1. If $T = \mathbf{x}$, then must have $U = \mathbf{x}$, by definition of $\trianglelefteq^{\maltese}_e$ and $\models_e$.
2. If $T = \mathtt{rec}\,\mathbf{x}.T'$, then by Lemma 5, we have
   (a) $U \models_{e,k} \boldsymbol{F}(T, \maltese) \iff U \models_{e,k} \boldsymbol{F}(T', \maltese)[{}^{\boldsymbol{F}(T,\maltese)}/_{\mathbf{x}}]$
   (b) $T \trianglelefteq^{\maltese}_{e,k} T'[{}^{T}/_{\mathbf{x}}] \trianglelefteq^{\maltese}_{e,k} T$
   Applying Lemma 1, it is enough to show that:

   $$\forall T, U \in \mathcal{T} : T'[{}^{\mathtt{rec}\,\mathbf{x}.T'}/_{\mathbf{x}}] \trianglelefteq^{\maltese}_{e,k} U \iff U \models_{e,k} \boldsymbol{F}(T'[{}^{\mathtt{rec}\,\mathbf{x}.T'}/_{\mathbf{x}}], \maltese)$$

   Hence, since we have assumed that the types are guarded, we only have to deal with the cases where $T = \maltese_{i \in I} \dagger a_i.\, T_i$, $T = \overline{\maltese}_{i \in I} \dagger a_i.\, T_i$, and $T = \mathtt{end}$. On the other hand, considering both sides of the equivalence (4), we notice that U cannot be a variable. Thus, let us assume that $U = \mathtt{rec}\,\mathbf{x}.U'$, by Lemma 5, we have
   (a) $U \models_{e,k} \boldsymbol{F}(T, \maltese) \iff U'[{}^{U}/_{\mathbf{x}}] \models_{e,k} \boldsymbol{F}(T, \maltese)$
   (b) $U \trianglelefteq^{\maltese}_{e,k} U'[{}^{U}/_{\mathbf{x}}] \trianglelefteq^{\maltese}_{e,k} U$
   Hence, applying Lemma 1 again, this case reduces to the cases where U is of the form: $\maltese_{j \in J} \dagger a_j.\, U_j$, $\overline{\maltese}_{j \in J} \dagger a_j.\, U_j$, or $\mathtt{end}$.
3. $T = \mathtt{end}$

- ($\Rightarrow$) Assume $\texttt{end} = T \trianglelefteq^{\maltese}_{e,k} U$, then by Definition 10, we have $U = \texttt{end}$. By Definition 2, we have $\boldsymbol{F}(\texttt{end}, \maltese) = [\mathcal{A}]\bot$, and we have $\texttt{end} \models_{e,k} [\mathcal{A}]\bot$ since $U = \texttt{end}\not\rightarrow$.
- ($\Leftarrow$) Assume $U \models_{e,k} \boldsymbol{F}(\texttt{end}, \maltese)$. By Definition 2, we have $U \models_{e,k} [\mathcal{A}]\bot$, which holds iff $U\not\rightarrow$, hence we must have $U = \texttt{end}$. Finally, by Definition 10, we have $\texttt{end} \trianglelefteq^{\maltese}_{e,k} \texttt{end}$.

4. $T = \maltese_{i\in I}\, \dagger a_i.\, T_i$
   - ($\Rightarrow$) Assume $\maltese_{i\in I}\, \dagger a_i.\, T_i \trianglelefteq^{\maltese}_{e,k} U$. By Definition 10, $U = \maltese_{j\in J}\, \dagger a_j.\, U_j$ with $I \subseteq J$ (note that $\varnothing \neq I$ by assumption) and $\forall i \in I : T_i \trianglelefteq^{\maltese}_{e,k-1} U_i$. Hence, $\forall i \in I : U \xrightarrow{\dagger a_i} U_i$, and by induction hypothesis, we have $U_i \models_{e,k-1} \boldsymbol{F}(T_i, \maltese)$, for all $i \in I$.
     By Definition 2, we have $\boldsymbol{F}(T, \maltese) = \bigwedge_{i\in I} \langle \dagger a_i \rangle \boldsymbol{F}(T_i, \maltese)$. Thus we have to show that for all $i \in I$, $U \xrightarrow{\dagger a_i} U_i$ and $U_i \models_{e,k-1} \boldsymbol{F}(T_i, \maltese)$; which follows from above.
   - ($\Leftarrow$) Assume $U \models_{e,k} \boldsymbol{F}(\maltese_{i\in I}\, \dagger a_i.\, T_i, \maltese)$. From Definition 2, we have

$$\boldsymbol{F}(T, \maltese) = \bigwedge_{i\in I} \langle \dagger a_i \rangle \boldsymbol{F}(T_i, \maltese)$$

     Hence, , $\forall i \in I : U \xrightarrow{\dagger a_i} U_i$, and $U_i \models_{e,k-1} \boldsymbol{F}(T_i, \maltese)$, for all $i \in I$. Hence, we must have $U = \maltese_{j\in J}\, \dagger a_j.\, U_j$ with $I \subseteq J$ and by induction hypothesis, this implies that $T_i \trianglelefteq^{\maltese}_{e,k-1} U_i$ for all $i \in I$.

5. $T = \overline{\maltese}_{i\in I}\, \dagger a_i.\, T_i$
   - ($\Rightarrow$) Assume $\overline{\maltese}_{i\in I}\, \dagger a_i.\, T_i \trianglelefteq^{\maltese}_{e,k} U$. By Definition 10, $U = \overline{\maltese}_{j\in J}\, \dagger a_j.\, U_j$, with $J \subseteq I$ and $\forall j \in J : T_j \trianglelefteq^{\maltese}_{e,k-1} U_j$. Hence, by induction hypothesis, we have $U_j \models_{e,k-1} \boldsymbol{F}(T_j, \maltese)$, for all $j \in J$.
     By Definition 2, we have

$$\boldsymbol{F}(T, \maltese) \;=\; \bigwedge_{i\in I} [\dagger a_i]\, \boldsymbol{F}(T_i, \maltese) \;\wedge\; \bigvee_{i\in I} \langle \dagger a_i \rangle \top \;\wedge\; [\neg\{\dagger a_i \mid i \in I\}]\bot \quad (5)$$

     We must show that $U \models_{e,k} \boldsymbol{F}(T, \maltese)$. Since $J \subseteq I$, we have that $\forall i \in I : T \xrightarrow{\dagger a_i} T_i \implies U \xrightarrow{\dagger a_i} U_i$, hence the first conjunct of (5) holds (using the induction hypothesis, cf. above). While the second conjunct of (5) must be true from the assumption that $\varnothing \neq J$. Finally, the third conjunct of (5) is false only if $U \xrightarrow{\dagger a_n}$ with $n \notin I$, which contradicts $J \subseteq I$.
   - ($\Leftarrow$) Assume $U \models_{e,k} \boldsymbol{F}(\overline{\maltese}_{i\in I}\, \dagger a_i.\, T_i, \maltese)$. From Definition 2, we have

$$\boldsymbol{F}(T, \maltese) \;=\; \bigwedge_{i\in I} [\dagger a_i]\, \boldsymbol{F}(T_i, \maltese) \;\wedge\; \bigvee_{i\in I} \langle \dagger a_i \rangle \top \;\wedge\; [\neg\{\dagger a_i \mid i \in I\}]\bot$$

     Hence, we must have $U = \overline{\maltese}_{j\in J}\, \dagger a_j.\, U_j$. It follows straightforwardly that $\varnothing \neq J \subseteq I$. Finally, the fact that for all $j \in J : T_j \trianglelefteq^{\maltese}_{e,k-1} U_j$, follows from the induction hypothesis. $\square$

A.4 Duality and safety in session types

Theorem 2. *For all* $T \in \mathcal{T}$: $\overline{\boldsymbol{F}(T,\maltese)} = \boldsymbol{F}(\overline{T},\overline{\maltese})$.

Proof. By straightforward induction on the structure of T.

1. The result follows trivially if $T = \mathtt{end}$ or $T = \mathbf{x}$.
2. If $T = \mathtt{rec}\,\mathbf{x}.T'$, then we have $\overline{\boldsymbol{F}(\mathtt{rec}\,\mathbf{x}.T',\maltese)} = \nu\mathbf{x}.\ \overline{\boldsymbol{F}(T',\maltese)}$, and $\boldsymbol{F}(\overline{T},\overline{\maltese}) = \nu\mathbf{x}.\ \boldsymbol{F}(\overline{T'},\overline{\maltese})$. The result follows by induction hypothesis.
3. If $T = \maltese_{i\in I} \dagger a_i.T_i$, then we have

$$\begin{aligned}
\overline{\boldsymbol{F}(T,\maltese)} &= \overline{\bigwedge_{i\in I}\langle\dagger\, a_i\rangle\, \boldsymbol{F}(T_i,\maltese)} \\
&= \bigwedge_{i\in I}\langle\overline{\dagger} a_i\rangle\, \overline{\boldsymbol{F}(T_i,\maltese)} \\
\text{(I.H.)} \quad &= \bigwedge_{i\in I}\langle\overline{\dagger} a_i\rangle\, \boldsymbol{F}(\overline{T_i},\overline{\maltese}) = \boldsymbol{F}(\overline{T},\overline{\maltese})
\end{aligned}$$

4. If $T = \overline{\maltese}_{i\in I} \dagger a_i.T_i$, then we have

$$\begin{aligned}
\overline{\boldsymbol{F}(T,\maltese)} &= \overline{\bigwedge_{i\in I}[\dagger\, a_i]\, \boldsymbol{F}(T_i,\maltese) \;\wedge\; \bigvee_{i\in I}\langle\dagger\, a_i\rangle\top \;\wedge\; [\neg\{\dagger\, a_i \mid i\in I\}]\bot} \\
&= \bigwedge_{i\in I}[\overline{\dagger} a_i]\, \overline{\boldsymbol{F}(T_i,\maltese)} \;\wedge\; \bigvee_{i\in I}\langle\overline{\dagger} a_i\rangle\top \;\wedge\; [\neg\{\overline{\dagger} a_i \mid i\in I\}]\bot \\
\text{(I.H.)} \quad &= \bigwedge_{i\in I}[\overline{\dagger} a_i]\, \boldsymbol{F}(\overline{T_i},\overline{\maltese}) \;\wedge\; \bigvee_{i\in I}\langle\overline{\dagger} a_i\rangle\top \;\wedge\; [\neg\{\overline{\dagger} a_i \mid i\in I\}]\bot \\
&= \boldsymbol{F}(\overline{T},\overline{\maltese})
\end{aligned}$$

□

Theorem 3 (Safety). $T \mid U$ *is safe* $\iff (T \leqslant \overline{U} \vee U \leqslant \overline{T})$.

Proof. ($\Longleftarrow$) We prove that if $T \leqslant \overline{U}$ then $T \mid U$ is safe by coinduction on the derivation of $T \leqslant \overline{U}$ (recall that $\leqslant$ stands for $\trianglelefteq^{\oplus}$).
Case [S-end] Obvious since $T = \overline{U} = \mathtt{end}$ and $T \mid U \not\rightarrow$.
Case [S-$\maltese$] Suppose $T = \bigoplus_{i\in I} !a_i.T_i$. Then $\overline{U} = \bigoplus_{j\in J} !a_j.\overline{U}_j$ such that $I \subseteq J$ and $T_i \leqslant \overline{U}_i$ for all $i \in I$. For all a_i such that $i \in I$, $T \xrightarrow{!a_i} T_i$ implies $U \xrightarrow{?a_i} U_i$. Hence by [S-COM], we have $T \mid U \rightarrow T_i \mid U_i$. Then by coinduction hypothesis, $T_i \mid U_i$ is safe.
Case [S-$\overline{\maltese}$] Similar to the above case.
($\Longrightarrow$) We prove $(\neg(T \leqslant \overline{U}) \wedge \neg(U \leqslant \overline{T}))$ implies $T \mid U$ has an error. Since the error rule coincides with the negation rules of subtyping in [9, Table 7], we conclude this direction. □

B Appendix: Recursive types for the λ-calculus

B.1 Recursive types and subtyping

We consider recursive types for the λ-calculus below:

$$t \coloneqq \texttt{top} \mid \texttt{bot} \mid t_0 \to t_1 \mid \texttt{rec}\, v.t \mid v$$

Let $\mathcal{T}_R$ be the set of all closed recursive types.

A type t induces an LTS according to the rules below:

$$\frac{}{\texttt{top} \xhookrightarrow{\texttt{top}} \texttt{top}}\ [\text{TOP}] \qquad \frac{}{\texttt{bot} \xhookrightarrow{\texttt{bot}} \texttt{top}}\ [\text{BOT}]$$

$$\frac{i \in \{0,1\}}{t_0 \to t_1 \xhookrightarrow{i} t_i}\ [\text{ARROW}] \qquad \frac{t[{}^{\texttt{rec}\, v.t}/_{v}] \xhookrightarrow{a} t'}{\texttt{rec}\, v.t \xhookrightarrow{a} t'}\ [\text{REC}]$$

where we let a range over $\{0, 1, \texttt{bot}, \texttt{top}\}$.

Definition 12 (Subtyping for recursive types). $\leqslant \subseteq \mathcal{T}_R \times \mathcal{T}_R$ *is the* largest *relation that contains the rules:*

$$\frac{t \in \mathcal{T}_R}{\texttt{bot} \leqslant t}\ [\text{S-BOT}] \qquad \frac{t \in \mathcal{T}_R}{t \leqslant \texttt{top}}\ [\text{S-TOP}] \qquad \frac{t_0' \leqslant t_0 \quad t_1 \leqslant t_1'}{t_0 \to t_1 \leqslant t_0' \to t_1'}\ [\text{S-ARROW}]$$

Recall that we are assuming an equi-recursive view of types. The double line in the rules indicates that the rules should be interpreted coinductively.

B.2 Characteristic formulae for recursive types

We assume the same fragment of the modal μ-calculus as in Section 3.1 but for (i) omitting the direction † on labels, i.e., we consider modalities: $[a]\phi$ and $\langle a\rangle\phi$; and (ii) using v to range over recursion variables.

Let $\delta \in \{\texttt{top}, \texttt{bot}\}$, and $\overline{\texttt{bot}} = \texttt{top}$, $\overline{\texttt{top}} = \texttt{bot}$.

$$\Lambda(t, \delta) \stackrel{\text{def}}{=} \begin{cases} \langle\delta\rangle\top & \text{if } t = \delta \\ \top & \text{if } t = \overline{\delta} \\ \langle 0\rangle\, \Lambda(t_0, \overline{\delta}) \ \wedge\ \langle 1\rangle\, \Lambda(t_1, \delta) & \text{if } t = t_0 \to t_1 \\ \nu v.\, \Lambda(t', \delta) & \text{if } t = \texttt{rec}\, v.t' \\ v & \text{if } t = v \end{cases}$$

Theorem 7. *The following holds:*

- $t \leqslant t' \iff t' \models \Lambda(t, \texttt{top})$
- $t \leqslant t' \iff t \models \Lambda(t', \texttt{bot})$

Proof. We show only the ($\Leftarrow$) direction here.

1. We show $t \leqslant t' \Leftarrow t' \models \Lambda(t, \texttt{top})$ by induction on t.
   - If $t = \texttt{top}$, then $\Lambda(t, \texttt{top}) = \langle\texttt{top}\rangle\top$, hence $t' = \texttt{top}$.
   - If $t = \texttt{bot}$, then $\Lambda(t, \texttt{top}) = \top$ hence t' can be any type, as expected.
   - If $t = t_0 \to t_1$, then

     $$\Lambda(t, \texttt{top}) = \langle 0\rangle\, \Lambda(t_0, \texttt{bot}) \;\wedge\; \langle 1\rangle\, \Lambda(t_1, \texttt{top})$$

     hence we must have $t' = t'_0 \to t'_1$ with $t'_0 \models \Lambda(t_0, \texttt{bot})$ (hence $t'_0 \leqslant t_0$, by IH, see below) and $t'_1 \models \Lambda(t_1, \texttt{top})$ (hence $t_1 \leqslant t'_1$ by IH).
2. We show $t \leqslant t' \Leftarrow t \models \Lambda(t', \texttt{bot})$ by induction on t'.
   - If $t' = \texttt{bot}$, then $\Lambda(t', \texttt{bot}) = \langle\texttt{bot}\rangle\top$ and $t = \texttt{bot}$.
   - if $t' = \texttt{top}$, then $\Lambda(t', \texttt{bot}) = \top$ and t can be any type, as expected.
   - If $t' = t'_0 \to t'_1$, then

     $$\Lambda(t', \texttt{bot}) = \langle 0\rangle\, \Lambda(t'_0, \texttt{top}) \;\wedge\; \langle 1\rangle\, \Lambda(t'_1, \texttt{bot})$$

     hence we must have $t = t_0 \to t_1$ with $t_0 \models \Lambda(t'_0, \texttt{top})$ (hence $t'_0 \leqslant t_0$, by IH, see above) and $t_1 \models \Lambda(t'_1, \texttt{bot})$ (hence $t_1 \leqslant t'_1$ by IH).

The other direction is similar to the above, while the recursive step is similar to the proof of Theorem 1. □