Intersections and Unions of Session Types

Coşku Acay
Carnegie Mellon University
Pennsylvania, USA
cacay@cmu.edu

Frank Pfenning
Carnegie Mellon University
Pennsylvania, USA
fp@cs.cmu.edu

Prior work has extended the deep, logical connection between the linear sequent calculus and session-typed message-passing concurrent computation with equi-recursive types and a natural notion of subtyping. In this paper, we extend this further by intersection and union types in order to express multiple behavioral properties of processes in a single type. We prove session fidelity and absence of deadlock and illustrate the expressive power of our system with some simple examples. We observe that we can represent internal and external choice by intersection and union, respectively, which was previously suggested in [6, 18] for a different language of session types motivated by operational rather than logical concerns.

1 Introduction

Prior work has established a Curry-Howard correspondence between intuitionistic linear sequent calculus and session-typed message-passing concurrency [5, 19, 17]. In this formulation, linear propositions are interpreted as session types, proofs as processes, and cut reduction as communication. Session types are assigned to channels and prescribe the communication behavior along them. Each channel is offered by a unique process and used by exactly one, which is ensured by linearity. When the behavior along a channel c satisfies the type A and P is the process that offers along c, we say that P provides a session of type A along c.

In the base system, each type directly corresponds to a process of a certain form. For example, a process providing the type $A \otimes B$ first sends out a channel satisfying A, then acts as B. Similarly, a process offering $\mathbf{1}$ sends the label `end` and terminates. We call these *structural types* since they correspond to processes of a certain structure. In this paper, we extend the base type system with intersections and unions. We call these *property types* since they do not correspond to specific forms of processes in that any process may be assigned such a type. In addition, if we interpret a type as specifying a property, then intersection corresponds to satisfying two properties simultaneously and union corresponds to satisfying one or the other.

Our goal is to show that the base system extended with intersection, unions, recursive types, and a natural notion of subtyping is type-safe. We do this by proving the usual type preservation and progress theorems, which correspond to session fidelity and deadlock freedom in the concurrent context. In the presence of a strong subtyping relation and transparent (i.e. non-generative) equi-recursive types, intersections and unions turn out to be powerful enough to specify many interesting communications behaviors, which we demonstrate with examples analogous to those in functional languages [12, 10].

Our contributions are summarized below:

- We introduce intersection and union types to a session-typed concurrent calculus and prove session fidelity and deadlock freedom.
- We give a simple and sound coinductive subtyping relation in the presence of equi-recursive types, intersections, and unions reminiscent of Gentzen's multiple conclusion sequent calculus [14, 15].

Submitted to:
ITRS 2016

- We show how intersections and unions can be used as refinements of recursive types in a linear setting.
- We show decidability of subtyping and present a system for algorithmic type checking.
- We demonstrate how internal and external choice can be understood as singletons interacting with intersection and union.

An extended version of this paper can be found at [1].

2 From Linear Logic to Session Types

We give only a brief review of linear logic and its connection to session types here. Interested readers are referred to [5, 19, 17]. The key idea of linear logic is to treat logical propositions as resources: each must be used exactly once in a proof. According to the Curry-Howard isomorphism for intuitionistic linear logic, propositions are interpreted as session types, proofs as concurrent processes, and cut-elimination steps as communication. For this correspondence, hypotheses are labelled with channels (rather than with variables). We also assign a channel name to the conclusion so that a process *providing* a session can refer to the same channel name as process *using* it. This replaces the usual notion of *duality* in classical session-typed calculi [17] and gives us the following form for typing judgments:

$$c_1 : A_1, \ldots, c_n : A_n \vdash P :: (c : A)$$

which should be interpreted as "*P provides along the channel c the session A using channels $c_1, \ldots, c_n$ (linearly) with their corresponding types*". We assume $c_1, \ldots, c_n$ and c are all distinct so that when a channel c occurs in P, its reference is unambiguous. We abbreviate hypotheses using Ψ, Ψ', etc.

Two key rules explaining this judgment are *cut* and *identity*. Logically, cut means that if we can prove A, then we can use it as a resource in the proof of some other proposition C. Operationally, it corresponds to *parallel composition* with a private shared variable for communication.

$$\frac{\Psi \vdash P_c :: (c : A) \quad \Psi', c : A \vdash Q_c :: (d : D)}{\Psi, \Psi' \vdash c \leftarrow P_c \,;\, Q_c :: (d : D)}\ \texttt{cut}$$

We write $c \leftarrow P_c \,;\, Q_c$ instead of the customary $(\nu c)(P_c \mid Q_c)$ because it is more readable in actual programs. The subscript here indicates that c is a bound variable and occurs in both P_c and Q_c.

Logically, the identity states the fundamental principle that the resource A can be used to prove the conclusion A. Operationally, it corresponds to *forwarding*: we provide a session along channel c by forwarding to d. Note that forwarding does not have a direct correspondence in the π-calculus but can be implemented in terms of other primitives (for example, $!(c(d).\overline{c}\langle d\rangle + d(c).\overline{d}\langle c\rangle)$).

$$\frac{}{d : A \vdash c \leftarrow d :: (c : A)}\ \texttt{id}$$

Forwarding is employed in session-typed programming with surprising frequency as will be evident from the examples we provide.

Cut and identity are general constructs, independent of any particular proposition. The isomorphism takes shape once we work out the interpretations of all of the connectives of linear logic as *session types*. We foreshadow the operational interpretation from the perspective of the provider of a session:

$$\begin{array}{lcll} A,B,C & ::= & \mathbf{1} & \text{send } \texttt{end} \text{ and terminate} \\ & | & A \otimes B & \text{send channel of type } A \text{ and continue as } B \\ & | & \oplus\{lab_k : A_k\}_{k\in I} & \text{send } lab_i \text{ and continue as } A_i \text{ for some } i \in I \\ & | & A \multimap B & \text{receive channel of type } A \text{ and continue as } B \\ & | & \&\{lab_k : A_k\}_{k\in I} & \text{receive } lab_i \text{ and continue as } A_i \text{ for some } i \in I \end{array}$$

In this paper, we do not need $!A$. Instead of replication, we use recursive types to describe recurring behavior. We also generalize the binary $A \oplus B$ to $\oplus\{lab_k : A_k\}_{k\in I}$ and $A\&B$ to $\&\{lab_k : A_k\}_{k\in I}$, which is in the tradition of much prior work on session types and makes programs more readable. Here, I is a *finite* non-empty set of distinct labels whose order does not matter. We call $\oplus\{lab_k : A_k\}_{k\in I}$ an internal choice since the provider selects which branch to take (i.e. the provider sends the label), and $\&\{lab_k : A_k\}_{k\in I}$ an external choice since the client makes the decision.

2.1 Process Expressions

The processes (or proof terms) corresponding to these types are given below with the sending construct followed by the receiving construct. The notation P_x emphasizes the scope of a bound variable x in the cut, send, and receive constructs. As a shorthand for substitution, we write P_a for $[a/x]P$, the capture-avoiding substitution of a for x in P.

$$\begin{array}{lcll} P,Q,R & ::= & x \leftarrow P_x \,;\, Q_x & \text{cut (spawn)} \\ & | & c \leftarrow d & \text{id (forward)} \\ & | & \text{close } c \mid \text{wait } c \,;\, P & \mathbf{1} \\ & | & \text{send } c\ (y \leftarrow P_y)\,;\, Q \mid x \leftarrow \text{recv } c \,;\, R_x & A \otimes B,\ A \multimap B \\ & | & c.lab \,;\, P \mid \text{case } c \text{ of } \{lab_k \to Q_k\}_{k\in I} & \&\{lab_k : A_k\}_{k\in I},\ \oplus\{lab_k : A_k\}_{k\in I} \end{array}$$

Because a process P always provides a session along a unique channel, we sometimes identify a process with the channel along which it provides. Keeping this in mind, the intuitive readings of process expressions are as follows. $x \leftarrow P_x \,;\, Q_x$ creates a fresh channel c, spawns a new process P_c providing along the channel c, and Q_c uses this new channel. The forwarding process $c \leftarrow d$ terminates by globally identifying channels c and d. Further communication along c will instead take place along d. A possible implementation of a forwarding process might tell the two end-points to communicate with each other and itself terminate, effectively, tying the two ends together and stepping out of the way. Process close c sends the token `end` and terminates, whereas the matching wait $c\,;\,P$ waits for `end` along c and continues as P. Processes send and recv are used for communicating channels along channels. In the case of send $c\ (y \leftarrow P_y)\,;\,Q$, a new channel d is created and a process P_d is spawned, but unlike cut, the continuation (Q) cannot refer to the new channel d, since it is sent along c to be used by a different process. Finally, $c.lab\,;\,P$ sends lab along c, and case c of $\{lab_k \to Q_k\}_{k\in I}$ receives a label along c and branches on it. These are used to determine which branches to take in internal and external choices.

2.2 Recursively Defined Types and Processes

For practical programming, we of course need recursive types and recursively defined processes. Usually, this is incorporated into the type language in a local way by introducing a new construct $\mu t.A_t$ and identifying $\mu t.A$ with its unfolding $[\mu t.A/t]A$ in the style of Amadio and Cardelli [2]. This makes it harder to incorporate mutual recursion, however, requiring type level pairs [21] for a fully formal treatment. We therefore go with the much simpler approach of using a global signature η of mutually recursive *type definitions* and *process definitions*. Type definitions are straightforward with the form $t = A$. They add defined type names as a new alternative to possible types. The fact that type definitions are transparent, that is, the fact that they are identified with their definition (or unfolding) without process-level terms to witness the isomorphism, corresponds to an equi-recursive interpretation of types. Process definitions have the form $X :: (c : A) = P_c$ where P provides session A along c.

$$\eta ::= \cdot \mid \eta, t = A \mid X :: (c : A) = P_c$$

Note that c is a bound variable with scope P. Formally, our judgments are now decorated with a fixed signature η, but we might elide it since the signature usually does not change during a derivation.

The identification of defined type names and their unfoldings is normally formalized by defining a new type equivalence judgment and adding conversion rules to process typing. In our formalization, we will integrate this into the subtyping judgment, but we delay its discussion to section 2.4. For now, it suffices to think of type definitions as finite representations of possibly infinite trees. For example, the definition $t = t \otimes t$ stands for the tree $(\ldots \otimes \ldots) \otimes (\ldots \otimes \ldots)$. However, for this to make sense, we need to slightly restrict valid type definitions to those that are contractive [21, 13]. In our setting, this means every type definition must have a structural type at the top. For example, the previous example of $t = t \otimes t$ is contractive since its top level construct is $\otimes$ whereas $t = t$ and $t = u$ are not.

As an example for signatures with recursion, consider processes producing a sequence of the label succ followed a single zero (i.e. Peano naturals).

$$\begin{array}{lcl}
\mathsf{Nat} & = & \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{Nat}\} \\
\mathsf{z} :: (c : \mathsf{Nat}) & = & c.\mathsf{zero} \,;\, \mathbf{close}\ c \\
\mathsf{s} :: (c : \mathsf{Nat} \multimap \mathsf{Nat}) & = & d \leftarrow \mathbf{recv}\ c \,;\, c.\mathsf{succ} \,;\, c \leftarrow d
\end{array}$$

The s process here receives a channel d representing a natural number, outputs one succ, and then behaves like d. Effectively, this computes the successor. To double a natural number, we need a recursive process definition.

```
double :: (c : Nat ⊸ Nat) =
  d ← recv c ;
  case d of
    zero → wait d ; c.zero ; close c
    succ → c.succ ; c.succ ;
            e ← double ; send e (d′ ← (d′ ← d)) ; c ← e
```

The last line here implements the following sequence:

$e \leftarrow \mathsf{double}$	cut: start a new process double which provides along the new channel e
send $e\ (d' \leftarrow (d' \leftarrow d))$	essentially, send d along e to the new process, but we have to use a forward
$c \leftarrow e$	now forward e to c

This is a frequent pattern, so we make two small improvements in the surface syntax: (1) we parameterize process definitions with the channels that they provide and use, and (2) instead of creating a new process and then forwarding it, we directly implement the provider channel with a new call. Then the definition becomes the much more readable

```
double : Nat ⊸ Nat
c ← double d =
  case d of
    zero → wait d ; c.zero ; close c
    succ → c.succ ; c.succ ;
            c ← double d
```

In an actual implementation, the new definition would be “desugared” to the previous one.

2.3 Type Assignment for Processes

The typing rules for processes are derived from linear logic by decorating derivations with proof terms. The rules are given in figure 1. Note that in $\oplus$L and &R, we allow unused branches in case expressions. This makes width subtyping easier, which is discussed in section 2.4. In addition, the def rule implicitly renames the channel name in the signature to the one expected by the judgment.

In our simple language, checking that a type is valid, $\vdash_\eta A : \mathsf{type}$, just verifies that all type names in A are defined in η and that A is contractive. A signature η itself is checked with the rules below. Note that we allow mutual recursion in the definitions which is witnessed by the fact that the signature η is propagated identically everywhere.

$$\frac{}{\vdash_\eta \emptyset} \qquad \frac{\vdash_\eta \eta' \quad \vdash_\eta A : \mathsf{type}}{\vdash_\eta \eta', t = A} \qquad \frac{\vdash_\eta \eta' \quad \vdash_\eta A : \mathsf{type} \quad \emptyset \vdash_\eta P_c :: (c : A)}{\vdash_\eta \eta', X :: (c : A) = P_c}$$

$$\frac{}{c : A \vdash d \leftarrow c :: (d : A)}\ \mathtt{id} \qquad \frac{\Psi \vdash P_c :: (c : A) \quad \Psi', c : A \vdash Q_c :: (d : D)}{\Psi, \Psi' \vdash c \leftarrow P_c \,;\, Q_c :: (d : D)}\ \mathtt{cut} \qquad \frac{}{\emptyset \vdash \mathrm{close}\ c :: (c : \mathbf{1})}\ \mathbf{1}\mathtt{R}$$

$$\frac{\Psi \vdash P :: (d : A)}{\Psi, c : \mathbf{1} \vdash \mathrm{wait}\ c ; P :: (d : A)}\ \mathbf{1}\mathtt{L} \qquad \frac{\Psi \vdash P_d :: (d : A) \quad \Psi' \vdash Q :: (c : B)}{\Psi, \Psi' \vdash \mathrm{send}\ c\ (d \leftarrow P_d) \,;\, Q :: (c : A \otimes B)}\ \otimes\mathtt{R}$$

$$\frac{\Psi, d : A, c : B \vdash P_d :: (e : E)}{\Psi, c : A \otimes B \vdash d \leftarrow \mathrm{recv}\ c ; P_d :: (e : E)}\ \otimes\mathtt{L} \qquad \frac{i \in I \quad \Psi \vdash P :: (c : A_i)}{\Psi \vdash c.lab_i ; P :: (c : \oplus\{lab_k : A_k\}_{k \in I})}\ \oplus\mathtt{R}$$

$$\frac{I \subseteq J \quad \Psi, c : A_k \vdash P_k :: (d : D) \text{ for } k \in I}{\Psi, c : \oplus\{lab_k : A_k\}_{k \in I} \vdash \mathrm{case}\ c \text{ of } \{lab_k \to P_k\}_{k \in J} :: (d : D)}\ \oplus\mathtt{L}$$

$$\frac{\Psi, d : A \vdash P_d :: (c : B)}{\Psi \vdash d \leftarrow \mathrm{recv}\ c ; P_d :: (c : A \multimap B)}\ \multimap\mathtt{R} \qquad \frac{\Psi \vdash P_d :: (d : A) \quad \Psi', c : B \vdash Q :: (e : E)}{\Psi, \Psi', c : A \multimap B \vdash \mathrm{send}\ c\ (d \leftarrow P_d) \,;\, Q :: (e : E)}\ \multimap\mathtt{L}$$

$$\frac{J \subseteq I \quad \Psi \vdash P_k :: (c : A_k) \text{ for } k \in J}{\Psi \vdash \mathrm{case}\ c \text{ of } \{lab_k \to P_k\}_{k \in I} :: (c : \&\{lab_k : A_k\}_{k \in J})}\ \&\mathtt{R}$$

$$\frac{i \in I \quad \Psi, c : A_i \vdash P :: (d : D)}{\Psi, c : \&\{lab_k : A_k\}_{k \in I} \vdash c.lab_i ; P :: (d : D)}\ \&\mathtt{L} \qquad \frac{X = P :: (c : A) \in \eta}{\emptyset \vdash X :: (d : A)}\ \mathsf{def}$$

Figure 1: Type assignment for process expressions

2.4 Subtyping

Gay and Hole [13] add coinductive subtyping (denoted $A \leq B$ in this paper) to their system in order to admit width and depth subtyping for n-ary choices, which are standard for record-like and variant-like structures. In our system, subtyping also doubles as a convenient way of identifying a recursive type and its unfolding using the following rules:

$$\frac{(t = B \in \eta) \quad A \leq_\eta B}{A \leq_\eta t}\ \mathsf{DefR} \qquad \frac{(t = A \in \eta) \quad A \leq_\eta B}{t \leq_\eta B}\ \mathsf{DefL}$$

Double lines here indicate that the rules should be interpreted coinductively as is common with theories using equi-recursive types. We will not go into the details of Gay and Hole's system since we will switch to a different relation in the next section anyway. Either way, we relate subtyping to process typing with subsumption rules:

$$\frac{\Psi \vdash_\eta P :: (c : A') \quad A' \leq_\eta A}{\Psi \vdash_\eta P :: (c : A)}\;\texttt{SubR} \qquad\qquad \frac{\Psi, c : A' \vdash_\eta P :: (d : B) \quad A \leq_\eta A'}{\Psi, c : A \vdash_\eta P :: (d : B)}\;\texttt{SubL}$$

2.5 Process Configurations

So far in the theory, we have only considered processes in isolation. In this section, we introduce process configurations in order to talk about the interactions between multiple processes. A process configuration, denoted by Ω, is simply a set of processes where each process is labelled with the channel along which it provides. We use the notation $\texttt{proc}_c(P)$ for labelling the process P, and require all labels in a configuration to be distinct.

With the above restriction, each process offers along a specific channel and each channel is offered by a unique process. Since channels are linear resources in our system, they must be used by exactly one process. In addition, we do not allow cyclic dependence, which imposes an implicit forest (set of trees) structure on a process configuration where each node has one outgoing edge and any number of incoming edges that correspond to channels the process uses. This observation suggests the typing rules below, which mimic the structure of a multi-way tree. Note that the definition is well founded since the size of the configuration gets strictly smaller.

$$\frac{}{\models \emptyset :: \emptyset}\;\texttt{config}_0 \qquad\qquad \frac{\models \Omega :: \Psi \quad \Psi \vdash P :: (c : A)}{\models \Omega, \texttt{proc}_c(P) :: (c : A)}\;\texttt{config}_1$$

$$\frac{\models \Omega_i :: (c_i : A_i) \text{ for } i \in \{1, \ldots, n\} \quad n > 1}{\models \Omega_1, \ldots, \Omega_n :: (c_1 : A_1, \ldots, c_n : A_n)}\;\texttt{config}_n$$

2.6 Operational Semantics

A process configuration evolves over time when a process takes a step, either by spawning a new process (cut), forwarding (id) or when two matching processes communicate. Our configurations are sets, so order is not significant when we match the left-hand sides against the configuration Ω. When we require a new name to be chosen, it must not already be offered by some process in the configuration. These rules are an example of a *substructural operational semantics* [20], presented in the form of a *multiset rewriting system* [7].

$$
\begin{array}{lcl}
\Omega, \mathtt{proc}_c(c \leftarrow d) & \longrightarrow & [d/c]\Omega \\
\Omega, \mathtt{proc}_c(x \leftarrow P_x \,;\, Q_x) & \longrightarrow & \Omega, \mathtt{proc}_a(P_a), \mathtt{proc}_c(Q_a) \quad (a \text{ fresh}) \\
\Omega, \mathtt{proc}_c(X) & \longrightarrow & \Omega, \mathtt{proc}_c([c/d]P) \quad (X = P :: (d : A) \in \eta) \\
\Omega, \mathtt{proc}_c(\text{close } c), \mathtt{proc}_e(\text{wait } c \,;\, P) & \longrightarrow & \Omega, \mathtt{proc}_e(P) \\
\Omega, \mathtt{proc}_c(\text{send } c\ (x \leftarrow P_x) \,;\, Q), \mathtt{proc}_e(x \leftarrow \text{recv } c \,;\, R_x) & \longrightarrow & \\
 & & \Omega, \mathtt{proc}_a(P_a), \mathtt{proc}_c(Q), \mathtt{proc}_e(R_a) \quad (a \text{ fresh}) \\
\Omega, \mathtt{proc}_c(c.lab_i \,;\, P), \mathtt{proc}_e(\text{case } c \text{ of } \{lab_k \rightarrow Q_k\}_{k \in I}) & \longrightarrow & \Omega, \mathtt{proc}_c(P), \mathtt{proc}_e(Q_i) \quad (i \in I) \\
\Omega, \mathtt{proc}_c(x \leftarrow \text{recv } c \,;\, P_x), \mathtt{proc}_d(\text{send } c\ (x \leftarrow Q_x) \,;\, R) & \longrightarrow & \\
 & & \Omega, \mathtt{proc}_c(P_a), \mathtt{proc}_a(Q_a), \mathtt{proc}_d(R) \quad (a \text{ fresh}) \\
\Omega, \mathtt{proc}_c(\text{case } c \text{ of } \{lab_k \rightarrow P_k\}_{k \in I}), \mathtt{proc}_e(c.lab_i \,;\, Q) & \longrightarrow & \Omega, \mathtt{proc}_c(P_i), \mathtt{proc}_e(Q) \quad (i \in I)
\end{array}
$$

This concludes the discussion of the base system. In the next section, we introduce intersections, unions, and a multiple-conclusion subtyping relation which constitute our main contributions.

3 Intersections and Unions

Recall our definition of process-level naturals Nat. One can imagine cases where we would like to know more about the exact nature of the natural. For example, if we are using a natural to track the size of a list, we might want to ensure it is non-zero. Sometimes, it might be relevant to track whether we have an even or an odd number. The system we have described so far turns out to be strong enough to describe all these *refinements* as illustrated below

$$
\begin{array}{lcl}
\mathsf{Nat} & = & \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{Nat}\} \\[1em]
\mathsf{Pos} & = & \oplus\{\mathsf{succ} : \mathsf{Nat}\} \\
\mathsf{Even} & = & \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{Odd}\} \\
\mathsf{Odd} & = & \oplus\{\mathsf{succ} : \mathsf{Even}\}
\end{array}
$$

Recall also the definitions

$$
\begin{array}{lll}
\mathsf{z} : \mathsf{Nat} & \mathsf{s} : \mathsf{Nat} \multimap \mathsf{Nat} & \mathsf{double} : \mathsf{Nat} \multimap \mathsf{Nat} \\
c \leftarrow \mathsf{z} = & c \leftarrow \mathsf{s}\ d = & c \leftarrow \mathsf{double}\ d = \\
\quad c.\mathsf{zero} \,;\, \textbf{close } c & \quad c.\mathsf{succ} \,;\, c \leftarrow d & \quad \textbf{case } d \textbf{ of} \\
 & & \qquad \mathsf{zero} \rightarrow \textbf{wait } d \,;\, c.\mathsf{zero} \,;\, \textbf{close } c \\
 & & \qquad \mathsf{succ} \rightarrow c.\mathsf{succ} \,;\, c.\mathsf{succ} \,; \\
 & & \qquad\qquad\qquad c \leftarrow \mathsf{double}\ d
\end{array}
$$

Intuitively, it is easy to see that Pos, Even, and Odd are all subtypes of Nat. We run into a problem when we try to implement the behavior described by these types, however. The s process, for example, satisfies many properties: Nat $\multimap$ Nat, Pos $\multimap$ Pos, Even $\multimap$ Odd, Odd $\multimap$ Even etc. Subtyping can be used to combine some of these (e.g. Nat $\multimap$ Pos for Nat $\multimap$ Nat and Pos $\multimap$ Pos) but it is not expressive enough to combine all properties. An elegant solution is to add intersections to the type system.

3.1 Intersection Types

We denote the intersection of two types A and B as $A \sqcap B$. A process offers an intersection type if its behavior satisfies both types simultaneously. Using intersections, we can assign the programs introduced in section 2.1 types specifying all behavioral properties we care about:

$$\begin{array}{lll} \mathsf{z} & : & \mathsf{Nat} \sqcap \mathsf{Even} \\ \mathsf{s} & : & (\mathsf{Nat} \multimap \mathsf{Nat}) \sqcap (\mathsf{Even} \multimap \mathsf{Odd}) \sqcap (\mathsf{Odd} \multimap \mathsf{Even}) \\ \mathsf{double} & : & (\mathsf{Nat} \multimap \mathsf{Nat}) \sqcap (\mathsf{Nat} \multimap \mathsf{Even}) \end{array}$$

Note that as is usual with intersections, multiple types are assigned to *the same process*. Put differently, we cannot use two different processes or specify two different behaviors to satisfy the different branches of an intersection. This leads to the following typing rule:

$$\frac{\Psi \vdash_\eta P :: (c : A) \quad \Psi \vdash_\eta P :: (c : B)}{\Psi \vdash_\eta P :: (c : A \sqcap B)} \sqcap \mathtt{R}$$

When we are using a channel on the left that offers an intersection of two types, we know it has to satisfy both properties so we get to pick the one we want:

$$\frac{\Psi, c : A \vdash_\eta P :: (d : D)}{\Psi, c : A \sqcap B \vdash_\eta P :: (d : D)} \sqcap \mathtt{L}_1 \qquad\qquad \frac{\Psi, c : B \vdash_\eta P :: (d : D)}{\Psi, c : A \sqcap B \vdash_\eta P :: (d : D)} \sqcap \mathtt{L}_2$$

It may seem as if the two left typing rules for intersection are somehow unnecessary: because of linearity, only one of A or B can be selected in any given derivation. But process definitions are used arbitrarily often, essentially spawning a new process along a new linear channel at each use point, so we may need to select a different component of the type at each occurrence. For example:

$$\begin{array}{ll} \mathsf{s} : (\mathsf{Even} \multimap \mathsf{Odd}) \sqcap (\mathsf{Odd} \multimap \mathsf{Even}) & \textit{(from before)} \\ \mathsf{s2} : \mathsf{Even} \multimap \mathsf{Even} & \\ c \leftarrow \mathsf{s2}\ d = & \\ \quad d_1 \leftarrow \mathsf{s}\ d & \textit{(use } \mathsf{s} : \mathsf{Even} \multimap \mathsf{Odd} \textit{ by } \sqcap L_1\textit{)} \\ \quad c \leftarrow \mathsf{s}\ d_1 & \textit{(use } \mathsf{s} : \mathsf{Odd} \multimap \mathsf{Even} \textit{ by } \sqcap L_2\textit{)} \end{array}$$

The standard subtyping rules are given below. It should be noted that the left typing rules above are derivable by an application of subsumption on the left using $\leq \sqcap \mathtt{L}_1$ and $\leq \sqcap \mathtt{L}_2$, so we will not explicitly add these to the final system. Also, we will have to modify the subtyping relation later in this section, so these rules are only a first attempt.

$$\frac{A \leq B_1 \qquad A \leq B_2}{A \leq B_1 \sqcap B_2} \leq \sqcap\mathtt{R} \qquad\qquad \frac{A_1 \leq B}{A_1 \sqcap A_2 \leq B} \leq \sqcap \mathtt{L}_1 \qquad\qquad \frac{A_2 \leq B}{A_1 \sqcap A_2 \leq B} \leq \sqcap \mathtt{L}_2$$

One final note about intersection types and recursion is that intersections are not considered structural types and thus do not contribute to contractiveness. That is, the type $t = t \sqcap t$ is *not* contractive.

3.2 Union Types

Unions are the dual of intersections and correspond to processes that satisfy one or the other property, and are written $A \sqcup B$. We add unions because they are a natural extension to a type system with intersections. We will also see how *n*-ary internal choice can be interpreted as the union of singleton choices. Without them, our interpretation would only be half-complete since we could interpret external choice (with intersections) but not internal choice.

Being dual to intersections, the typing rules for unions mirror the typing rules for intersections: we have two right rules and one left rule, and this time the right rules are derivable from subtyping. The rules are given below:

$$\frac{\Psi \vdash_\eta P :: (c : A)}{\Psi \vdash_\eta P :: (c : A \sqcup B)} \sqcup \mathtt{R}_1 \qquad \frac{\Psi \vdash_\eta P :: (c : B)}{\Psi \vdash_\eta P :: (c : A \sqcup B)} \sqcup \mathtt{R}_2$$

$$\frac{\Psi, c : A \vdash_\eta P :: (d : D) \quad \Psi, c : B \vdash_\eta P :: (d : D)}{\Psi, c : A \sqcup B \vdash_\eta P :: (d : D)} \sqcup \mathtt{L}$$

The right rules state the process has to offer either the left type or the right type respectively. The left rule says we need to be prepared to handle either type. It is interesting to observe that the usual problems with unions in functional languages do not arise in our setting. The natural left rule we give here (natural since it is dual to the right rule for intersection) has been shown to be unsound in functional languages [3]. One somewhat heavy solution limits the left rule to expressions in evaluation position [11]. The straightforward left rule turns out to be already sound here, essentially due to linearity and the use of the sequent calculus.

The usual subtyping rules are given below. The $\sqcup \mathtt{R}$ rules are derivable by general subtyping, so they don't need to be explicitly added to the system.

$$\frac{A \le B_1}{A \le B_1 \sqcup B_2} \le\sqcup \mathtt{R}_1 \qquad \frac{A \le B_2}{A \le B_1 \sqcup B_2} \le\sqcup \mathtt{R}_1 \qquad \frac{A_1 \le B \quad A_2 \le B}{A_1 \sqcup A_2 \le B} \le\sqcup \mathtt{L}$$

For an example where unions are the most natural way to express a property, consider moving to a binary representation of natural numbers. We define the type of binary string where the least significant bit is sent first and the string of bits is terminated with eps:

$$\mathsf{Bits} \;=\; \oplus\{\mathsf{eps} : \mathbf{1}, \mathsf{zero} : \mathsf{Bits}, \mathsf{one} : \mathsf{Bits}\}$$

We can define bit strings in standard form (no leading zeros) as follows:

$$\begin{array}{lcl} \mathsf{Std} & = & \mathsf{Empty} \sqcup \mathsf{StdPos} \\ \mathsf{Empty} & = & \oplus\{\mathsf{eps} : \mathbf{1}\} \\ \mathsf{StdPos} & = & \oplus\{\mathsf{one} : \mathsf{Std}, \mathsf{zero} : \mathsf{StdPos}\} \end{array}$$

We are able to naturally express standard bit strings as either an empty bit string *or* a positive one; expressing such types without unions would be cumbersome at the very least.[1] Now we can write an

[1]The acute reader might notice that Std actually fails our simple contractiveness criteria. This does not lead to unsoundness, however, since its one level unfolding is contractive (i.e. inlining Empty and StdPos makes it contractive). We write it in closed form here for better readability, but one can consider this simple syntactic sugar. It is not too hard to formulate a sound contractiveness condition that allows such definitions, but we decided to err on the side of simplicity for this paper.

increment function that preserves bit strings in standard form:

```
$\mathsf{inc} : \mathsf{Std} \multimap \mathsf{Std} \sqcap \mathsf{StdPos} \multimap \mathsf{StdPos} \sqcap \mathsf{Empty} \multimap \mathsf{StdPos}$

$c \leftarrow \mathsf{inc}\ d =$
  case $d$ of
    $\mathsf{eps} \rightarrow$ wait $d$ ; $c.\mathsf{one}$ ; $c.\mathsf{eps}$ ; close $c$
    $\mathsf{zero} \rightarrow c.\mathsf{one}$ ; $c \leftarrow d$
    $\mathsf{one} \rightarrow c.\mathsf{zero}$ ; $\mathsf{inc}\ d$
```

This example also demonstrates that for a recursively defined type we may need to specify more information than we ultimately care about, since checking this definition just against the type $\mathsf{Std} \multimap \mathsf{Std}$ will fail, and we need to assign the more specific type for the type checking to go through. This is because of the nature of our system which essentially requires the type checker to verify a fixed point rather than infer the least one. This has proven highly beneficial for providing good error messages even in the simpler case of pure subtyping, without intersections and unions [16].

3.3 Subtyping Revisited

In line with our propositional interpretation of intersections and unions, one would naturally expect the usual properties of these to hold in our system. For example, unions should distribute over intersections and vice versa, that is, the following equalities should be admissible:

$$(A_1 \sqcup B) \sqcap (A_2 \sqcup B) \equiv (A_1 \sqcap A_2) \sqcup B$$

$$(A_1 \sqcup A_2) \sqcap B \equiv (A_1 \sqcap B) \sqcup (A_2 \sqcap B)$$

Going from right to left turns out to be easy, but we quickly run into a problem if we try to do the other direction: whether we break down the union on the right or the intersection on the left, we always lose half the information we need to carry out the rest of the proof.[2]

Our solution is doing the obvious: if the problem is losing half the information, well, we should just keep it around. This suggests a system where the single type on the left and the type on the right are replaced with *(multi)sets* of types. That is, instead of the judgment $A \leq B$, we use a judgment of the form $A_1, \ldots, A_n \Rightarrow B_1, \ldots, B_n$, where the left of $\Rightarrow$ is interpreted as a conjunction (intersection) and the right is interpreted as a disjunction (union).[3] This results in a system reminiscent of [14, 15]. However, we take a slightly different approach since we are working with coinductive rules.

The rules are given in figure 2. We use α and β to denote multisets of types. The intersection left rules are combined into one rule that keeps both branches around. The same is done with union right rules. Intersection right and union left rules split into two derivations, one for each branch, but keep the rest of the types unchanged. We can unfold a recursive type on the left or on the right. When we choose to apply a structural rule, we have to pick exactly one type on the left and one on the right with the same structure. We conjecture that matching multiple types might give us distributivity of intersection and union over structural types, although a naive extension along these lines fails to satisfy type safety.

[2]This issue does not come up in the other direction since intersection right and union left rules are invertible, that is, they preserve all information.

[3]We use multisets rather than sets since types have nontrivial equality, so it is not obvious when we should combine them into one.

$$\frac{\alpha \Rightarrow \beta, A_1 \qquad \alpha \Rightarrow \beta, A_2}{\alpha \Rightarrow \beta, A_1 \sqcap A_2} \Rightarrow \sqcap\mathsf{R} \qquad\qquad \frac{\alpha, A_1, A_2 \Rightarrow \beta}{\alpha, A_1 \sqcap A_2 \Rightarrow \beta} \Rightarrow \sqcap\mathsf{L}$$

$$\frac{\alpha \Rightarrow \beta, A_1, A_2}{\alpha \Rightarrow \beta, A_1 \sqcup A_2} \Rightarrow \sqcup\mathsf{R} \qquad\qquad \frac{\alpha, A_1 \Rightarrow \beta \quad \alpha, A_2 \Rightarrow \beta}{\alpha, A_1 \sqcup A_2 \Rightarrow \beta} \Rightarrow \sqcup\mathsf{L}$$

$$\frac{}{\alpha, \mathbf{1} \Rightarrow \beta, \mathbf{1}} \Rightarrow \mathbf{1} \qquad \frac{A \Rightarrow A' \quad B \Rightarrow B'}{\alpha, A \otimes B \Rightarrow \beta, A' \otimes B'} \Rightarrow \otimes \qquad \frac{I \subseteq J \quad A_k \Rightarrow A'_k \text{ for } k \in I}{\alpha, \oplus\{lab_k : A_k\}_{k \in I} \Rightarrow \beta, \oplus\{lab_k : A'{}_k\}_{k \in J}} \Rightarrow \oplus$$

$$\frac{A' \Rightarrow A \quad B \Rightarrow B'}{\alpha, A \multimap B \Rightarrow \beta, A' \multimap B'} \Rightarrow \multimap \qquad\qquad \frac{J \subseteq I \quad A_k \Rightarrow A'_k \text{ for } k \in J}{\alpha, \&\{lab_k : A_k\}_{k \in I} \Rightarrow \beta, \&\{lab_k : A'{}_k\}_{k \in J}} \Rightarrow \&$$

$$\frac{(t = A \in \eta) \quad \alpha \Rightarrow \beta, A}{\alpha \Rightarrow \beta, t} \Rightarrow \mathsf{DefR} \qquad\qquad \frac{(t = A \in \eta) \quad \alpha, A \Rightarrow \beta}{\alpha, t \Rightarrow \beta} \Rightarrow \mathsf{DefL}$$

Figure 2: Subtyping with multiple hypothesis and conclusions; coinductively with respect to a fixed signature η

3.4 Reinterpreting Choice

In this section, we show that that intersections and unions are useful beyond their refinement interpretation, and help us understand external and internal choices better. Take external choice, for instance. A comparison between the typing rules for intersections and external choice reveal striking similarities. The only difference, in fact, is that internal choice has process-level constructs whereas intersections are implicit.

Consider the special case of binary external choice: $\&\{\mathsf{inl} : A, \mathsf{inr} : B\}$. This type says: I will act as A if you send me inl *and* I will act as B if you send me inr. We know the *and* can be interpreted as an intersection, and either side can be thought of as a singleton internal choice. A similar argument can be given for internal choice and unions. This gives us the following redefinitions of n-ary external and internal choices:

$$\&\{lab_k : A_k\}_{k \in I} \triangleq \bigsqcap_{k \in I} \&\{lab_k : A_k\}$$

$$\oplus\{lab_k : A_k\}_{k \in I} \triangleq \bigsqcup_{k \in I} \oplus\{lab_k : A_k\}$$

It is a straightforward calculation that these definitions satisfy the typing and subtyping rules for external and internal choices.

4 Algorithmic System

In this section, we show that subtyping and type-checking are decidable by designing an algorithm that takes in a (sub)typing judgment and produces true if and only if there is a derivation. Note that everything in the judgment is considered an input.

4.1 Algorithmic Subtyping

The subtyping judgment we gave is already mostly algorithmic (a necessity of working with coinductive rules), so we only have to tie up a few loose ends. The first is deciding which rule to pick when multiple are applicable. We apply $\Rightarrow \sqcap\,\mathtt{R}$, $\Rightarrow \sqcap\,\mathtt{L}$, $\Rightarrow \sqcup\,\mathtt{R}$, $\Rightarrow \sqcup\,\mathtt{L}$, $\Rightarrow \mathtt{DefR}$, $\Rightarrow \mathtt{DefL}$ eagerly since these are invertible. At some point, all types must be structural (since definitions are restricted to be contractive), at which point we non-deterministically pick a structural rule and continue. In the implementation, we backtrack over these choices.

Second, the coinductive nature of subtyping means we can (and often will) have infinite derivations. We combat this by using a cyclicity check (similar to the one in [13]): we maintain a context of previously seen subtyping comparisons and immediately terminate with success if we ever compare the same pair of sets of types again. Every recursive step corresponds to a rule, which ensures a productive derivation. We know there cannot be an infinite chain of new types due to the contractiveness restriction which implies an upper-bound on the size of the previously-seen set. A more formal treatment can be found in [21].

4.2 Algorithmic Type-checking

Designing a type checking algorithm is quite simple for the base system where we only have structural types (no recursive types or subtyping), since the form of the process determines a unique applicable typing rule. The `cut` rule causes a small problem since we do not have a type for the new channel. This is solved by requiring a type annotation when necessary such that the new term becomes $c : A \leftarrow P_c\,;\,Q_c$. The overwhelmingly common case where it is *not* necessary is when P_c is a defined process name X because we simply fall back on its given type.

In the extended system with subtyping and property types, type-checking is trickier for two reasons: (1) subsumption can be applied anytime where one of the types in $A \le B$ can be anything (the other will be fixed due to typing rules, but one causes enough damage), and (2) intersection left and union right rules lose information which means they have to be applied non-deterministically. The latter issue is resolved by switching to a judgment where each channel (whether on the left or the right) is assigned a multiset of types. These multisets are interpreted conjunctively for channels used (on the left) and disjunctively for the channel provided (on the right). This makes intersection left and union right rules invertible, so they can be applied eagerly.

The former problem is solved by checking subtyping only at the identity rule (forwarding). This relies on the subformula property for the sequent calculus, excepting only the cut rule which is annotated. The new judgment is written $\Psi \Vdash_\eta P :: (c : \alpha)$, where Ψ and α are multisets. Typing rules are given in figure 3. Note the explicit $\sqcap\,\mathtt{L}$, $\sqcup\,\mathtt{R}$, $\mathtt{DefL}$, and $\mathtt{DefR}$. These rules were derivable in the declarative system using subsumption, which is no longer possible since application of subsumption is restricted to forwarding processes.

4.3 Equivalence to the Declarative System

Next, we show that the algorithmic system is sound and complete with respect to the declarative system (modulo erasure of type annotations, which we denote by $[\![P]\!]$). Due to space limitations, we can only give very brief proof sketches here. Interested readers are referred to the first author's thesis [1]. We define $\bigsqcup \alpha$ as the union of all the types in α, and similarly for $\bigsqcap \alpha$. For contexts we define $\bigsqcap(c_1{:}\alpha_1, \ldots, c_n{:}\alpha_n) = c_1{:}\bigsqcap \alpha_1, \ldots, c_n{:}\bigsqcap \alpha_n$.

Theorem 4.1 (Soundness of Algorithmic Typing)**.** *If* $\Psi \Vdash_\eta P :: (c : \alpha)$*, then* $\bigsqcap \Psi \vdash_\eta [\![P]\!] :: (c : \bigsqcup \alpha)$.

$$\frac{\Psi \Vdash_\eta P :: (c : A, \alpha) \quad \Psi \Vdash_\eta P :: (c : B, \alpha)}{\Psi \Vdash_\eta P :: (c : A \sqcap B, \alpha)}\ \sqcap\mathtt{R} \qquad \frac{\Psi, c : (\alpha, A, B) \Vdash_\eta P :: (d : \beta)}{\Psi, c : (\alpha, A \sqcap B) \Vdash_\eta P :: (d : \beta)}\ \sqcap\mathtt{L}$$

$$\frac{\Psi \Vdash_\eta P :: (c : A, B, \alpha)}{\Psi \Vdash_\eta P :: (c : A \sqcup B, \alpha)}\ \sqcup\mathtt{R} \qquad \frac{\Psi, c : (\alpha, A) \Vdash_\eta P :: (d : \beta) \quad \Psi, c : (\alpha, B) \Vdash_\eta P :: (d : \beta)}{\Psi, c : (\alpha, A \sqcup B) \Vdash_\eta P :: (d : \beta)}\ \sqcup\mathtt{L}$$

$$\frac{(t = A \in \eta) \quad \Psi \Vdash_\eta P :: (c : A, \alpha)}{\Psi \Vdash_\eta P :: (c : t, \alpha)}\ \mathsf{Def}\mathtt{R} \qquad \frac{(t = A \in \eta) \quad \Psi, c : (\alpha, A) \Vdash_\eta P :: (d : \beta)}{\Psi, c : (\alpha, t) \Vdash_\eta P :: (d : \beta)}\ \mathsf{Def}\mathtt{L}$$

$$\frac{\alpha \Rightarrow \beta}{c : \alpha \Vdash_\eta d \leftarrow c :: (d : \beta)}\ \mathtt{id} \qquad \frac{\Psi \Vdash_\eta P_c :: (c : A) \quad \Psi', c : A \Vdash_\eta Q_c :: (d : \alpha)}{\Psi, \Psi' \Vdash_\eta c : A \leftarrow P_c \,;\, Q_c :: (d : \alpha)}\ \mathtt{cut}$$

$$\frac{}{\emptyset \Vdash_\eta \text{close}\ c :: (c : \mathbf{1}, \alpha)}\ \mathbf{1}\mathtt{R} \qquad \frac{\Psi \Vdash_\eta P :: (d : \beta)}{\Psi, c : (\alpha, \mathbf{1}) \Vdash_\eta \text{wait}\ c \,;\, P :: (d : \beta)}\ \mathbf{1}\mathtt{L}$$

$$\frac{\Psi \Vdash_\eta P :: (d : A) \quad \Psi' \Vdash_\eta Q :: (c : B)}{\Psi, \Psi' \Vdash_\eta \text{send}\ c\ (d \leftarrow P_d)\,;\, Q :: (c : A \otimes B, \alpha)}\ \otimes\mathtt{R} \qquad \frac{\Psi, d : A, c : B \Vdash_\eta P_d :: (e : \beta)}{\Psi, c : (\alpha, A \otimes B) \Vdash_\eta d \leftarrow \text{recv}\ c \,;\, P_d :: (e : \beta)}\ \otimes\mathtt{L}$$

$$\frac{i \in I \quad \Psi \Vdash_\eta P :: (c : A_i)}{\Psi \Vdash_\eta c.lab_i \,;\, P :: (c : \oplus\{lab_k : A_k\}_{k \in I}, \alpha)}\ \oplus\mathtt{R}$$

$$\frac{I \subseteq J \quad \Psi, c : A_k \Vdash_\eta P_k :: (d : \beta)\ \text{for}\ k \in I}{\Psi, c : (\alpha, \oplus\{lab_k : A_k\}_{k \in I}) \Vdash_\eta \text{case}\ c\ \text{of}\ \{lab_k \to P_k\}_{k \in J} :: (d : \beta)}\ \oplus\mathtt{L}$$

$$\frac{\Psi, d : A \Vdash_\eta P_d :: (c : B)}{\Psi \Vdash_\eta d \leftarrow \text{recv}\ c \,;\, P_d :: (c : A \multimap B, \alpha)}\ \multimap\mathtt{R}$$

$$\frac{\Psi \Vdash_\eta P_d :: (d : A) \quad \Psi', c : B \Vdash_\eta Q :: (e : \beta)}{\Psi, \Psi', c : (\alpha, A \multimap B) \Vdash_\eta \text{send}\ c\ (d \leftarrow P_d)\,;\, Q :: (e : \beta)}\ \multimap\mathtt{L}$$

$$\frac{J \subseteq I \quad \Psi \Vdash_\eta P_k :: (c : A_k)\ \text{for}\ k \in J}{\Psi \Vdash_\eta \text{case}\ c\ \text{of}\ \{lab_k \to P_k\}_{k \in I} :: (c : \&\{lab_k : A_k\}_{k \in J}, \alpha)}\ \&\mathtt{R}$$

$$\frac{i \in I \quad \Psi, c : A_i \Vdash_\eta P :: (d : \beta)}{\Psi, c : (\alpha, \&\{lab_k : A_k\}_{k \in I}) \Vdash_\eta c.lab_i \,;\, P :: (d : \beta)}\ \&\mathtt{L} \qquad \frac{X = P :: (c : A) \in \eta}{\Psi \Vdash_\eta \emptyset :: (X : d)A}\ \mathsf{def}$$

Figure 3: Process Typing in the Algorithmic System

Proof. By induction on the typing derivation. The only non-straightforward cases are $\sqcap\,\mathtt{R}$ and $\sqcup\,\mathtt{L}$, which depend on the distributivity of intersection and union over each other (which is one of the reasons why we insisted such be the case while designing the subtyping relation). □

Lemma 4.2 (Completeness of Delayed Subtyping)**.** *The following are admissible:*

- *If* $\Psi \Vdash_\eta P :: (c : \alpha)$ *and* $\bigsqcup \alpha \Rightarrow \beta$ *then* $\Psi \Vdash_\eta P :: (c : \beta)$*.*

- *If* $\Psi, d : \alpha \Vdash_\eta P :: (c : \beta)$ *and* $\alpha' \Rightarrow \bigsqcap \alpha$ *then* $\Psi, d : \alpha' \Vdash_\eta P :: (c : \beta)$.

Note that the type annotations in P stay the same.

Proof. By lexicographic induction, first on the structure of P, then on the combined sizes of involved types. □

Theorem 4.3 (Completeness of Algorithmic Typing)**.** *If* $\Psi \vdash_\eta P :: (c : A)$*, then there exists* P' *such that* $[\![P']\!] = P$ *and* $\Psi \Vdash_\eta P' :: (c : A)$.

Proof. By induction on the typing derivation, using lemma 4.2 for `SubR` and `SubL`. □

5 Metatheory

Our main contribution is proving type safety for the system with intersections and unions, which we do so by showing the standard progress and preservation theorems, renamed to deadlock freedom and session fidelity, respectively, within this context. Since the algorithmic system is more well behaved (no subsumption), we use the algorithmic judgment in the statements and proofs of these results. Type safety for the declarative system follows from its equivalence to the algorithmic system. We only state the theorems here. Full proofs can be found in [1].

In a functional setting, progress states a well-typed expression either takes a step or is a value. The corresponding notion of a value is a *poised* configuration. A configuration is poised if every process in it is, and a process is poised if it is waiting to communicate with its client. With this definition, we can state the progress theorem:

Theorem 5.1 (Progress)**.** *If* $\models \Omega :: \Psi$ *then either*

1. $\Omega \longrightarrow \Omega'$ *for some* Ω'*, or*
2. Ω *is poised.*

Proof. By induction on $\models \Omega :: \Psi$ followed by a nested induction on the typing of the root process for the $\texttt{config}_1$ case. When two processes are involved, we also need inversion on client's typing. □

Theorem 5.2 (Preservation)**.** *If* $\models \Omega :: \Psi$ *and* $\Omega \longrightarrow \Omega'$ *then* $\models \Omega' :: \Psi$.

Proof. By inversion on $\Omega \longrightarrow \Omega'$, followed by induction on the typing judgments of the involved processes. □

6 Related Work and Conclusion

Padovani describes a calculus similar to ours [18] where he interprets internal and external choices as union and intersection, respectively. This resembles what we did in section 3.4 except we keep singleton choices at the type and term levels to maintain the connection to linear logic, whereas Padovani is able to remove them completely since his calculus is based on CSS [8]. While we give axiomatic rules for (sub)typing, he takes a semantic approach, which we believe is complementary to our work. However, semantic definitions make deriving algorithmic rules harder and he leaves this as future work. More significantly, Padovani does not consider higher-order types and processes, which means it is not possible to communicate channels along channels. Moreover, his calculus only deals with the interaction between two processes that are required to have matching (or dual) types and behaviors (for example, if one sends, the other must receive etc.). We consider a tree of processes where each process can use many providers (as long as it respect the behavior along their channels) and even spawn new ones. For example, a client using two providers could communicate with one of them while the other is idle, or ignore both altogether and only communicate with *its* client.

Castagna et al. describe a generic framework [6] that make use of set operations (intersection, union, and negation) for sessions and take a semantic approach as well. Their framework has the advantage that it is agnostic to the underlying functional language. There are descriptions of our base calculus that take a similar approach [22] and it should not be too hard to extend them to cover our contributions. Contrary to Padovani, Castagna et al. have higher order sessions and give algorithms to decide all semantic relations they describe. However, their system and presentation are significantly different from ours because of their semantic emphasis, inclusion of negation (which makes their session language Boolean), and treatment of process composition (which is closer to Padovani's). In particular, they do not have a general primitive for spawning new processes or forwarding.

We introduced intersections and unions to a simple system of session types, and demonstrated how they can be used to refine behavioral specifications of processes. Some aspects that would be important in a full accounting of the system are omitted for the sake of brevity or are left as future work. For example, integrating an underlying functional language [22], adding shared channels [5, 19], or considering asynchronous communication [9, 19, 16] are straightforward extensions based on prior work. In addition, it would be very useful to have behavioral polymorphism [4] and abstract types. Their interaction with subtyping, intersections, and unions is an interesting avenue for future work.

Acknowledgments. This work was funded in part by NSF grant CNS1423168 and by the FCT (Portuguese Foundation for Science and Technology) through the Carnegie Mellon Portugal Program. We would like to thank the anonymous referees for their many helpful suggestions.

References

[1] Coşku Acay (2016): *Refinements for Session Typed Concurrency*. Undergraduate honors thesis, Carnegie Mellon University. Available at `http://www.coskuacay.com/papers/senior-thesis.pdf`.

[2] Roberto M. Amadio & Luca Cardelli (1991): *Subtyping Recursive Types*. In: *POPL*, ACM Press, pp. 104–118.

[3] Franco Barbanera, Mariangiola Dezani-Ciancaglini & Ugo de'Liguoro (1995): *Intersection and Union Types: Syntax and Semantics*. *Information and Computation* 119, pp. 202–230.

[4] Luís Caires, Jorge A. Pérez, Frank Pfenning & Bernardo Toninho (2013): *Behavioral Polymorphism and Parametricity in Session-Based Communication*. In: *ESOP*, *Lecture Notes in Computer Science* 7792, Springer, pp. 330–349.

[5] Luís Caires & Frank Pfenning (2010): *Session Types as Intuitionistic Linear Propositions*. In: *CONCUR*, *Lecture Notes in Computer Science* 6269, Springer, pp. 222–236.

[6] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, Elena Giachino & Luca Padovani (2009): *Foundations of session types*. In: *PPDP*, ACM, pp. 219–230.

[7] Iliano Cervesato & Andre Scedrov (2009): *Relating state-based and process-based concurrency through linear logic (full-version)*. *Inf. Comput.* 207(10), pp. 1044–1077.

[8] Rocco De Nicola & Matthew Hennessy (1987): *CCS without tau's*. In: *TAPSOFT, Vol.1*, *Lecture Notes in Computer Science* 249, Springer, pp. 138–152.

[9] Henry DeYoung, Luís Caires, Frank Pfenning & Bernardo Toninho (2012): *Cut Reduction in Linear Logic as Asynchronous Session-Typed Communication*. In: *CSL*, *LIPIcs* 16, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 228–242.

[10] Joshua Dunfield & Frank Pfenning (2003): *Type Assignment for Intersections and Unions in Call-by-Value Languages*. In: *FoSSaCS*, *Lecture Notes in Computer Science* 2620, Springer, pp. 250–266.

[11] Joshua Dunfield & Frank Pfenning (2004): *Tridirectional typechecking*. In: *POPL*, ACM, pp. 281–292.

[12] Tim Freeman & Frank Pfenning (1991): *Refinement Types for ML*. In: *PLDI*, ACM, pp. 268–277.

[13] Simon J. Gay & Malcolm Hole (2005): *Subtyping for session types in the pi calculus*. *Acta Inf.* 42(2-3), pp. 191–225.

[14] Gerhard Gentzen (1935): *Untersuchungen über das Logische Schließen*. *Mathematische Zeitschrift* 39, pp. 176–210, 405–431. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.

[15] Jean-Yves Girard (1987): *Linear Logic*. *Theor. Comput. Sci.* 50, pp. 1–102.

[16] Dennis Griffith (2016): *Polarized Substructural Session Types*. Ph.D. thesis, University of Illinois at Urbana-Champaign. In preparation.

[17] Kohei Honda (1993): *Types for Dyadic Interaction*. In: *CONCUR*, *Lecture Notes in Computer Science* 715, Springer, pp. 509–523.

[18] Luca Padovani (2010): *Session Types = Intersection Types + Union Types*. In: *ITRS*, *EPTCS* 45, pp. 71–89.

[19] Frank Pfenning & Dennis Griffith (2015): *Polarized Substructural Session Types*. In: *FoSSaCS*, *Lecture Notes in Computer Science* 9034, Springer, pp. 3–22.

[20] Robert J. Simmons (2012): *Substructural Logical Specifications*. Ph.D. thesis, Carnegie Mellon University.

[21] Christopher A. Stone & Andrew P. Schoonmaker (2005): *Equational Theories with Recursive Types*. Unpublished Manuscript.

[22] Bernardo Toninho, Luís Caires & Frank Pfenning (2013): *Higher-Order Processes, Functions, and Sessions: A Monadic Integration*. In: *ESOP*, *Lecture Notes in Computer Science* 7792, Springer, pp. 350–369.