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 linear sequent calculus and session-typed message-passing concurrent computation with equirecursive 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, 17] 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, 18, 16]. In this formulation, linear propositions are interpreted as session types, proofs as processes, and cut-elimination 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) equirecursive 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 [11, 9].

Our contributions are summarized below:

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

[†]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.

Submitted to:
ITRS 2016

- 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 equirecursive types, intersections, and unions reminiscent of Gentzen's multiple conclusion sequent calculus [13, 14].
- 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.

2 From Linear Logic to Session Types

We only give a brief review of linear logic and its connection to session types here. Interested readers are referred to [5, 18, 16]. The key idea of linear logic is to treat logical propositions as resources: each must be used *exactly* once. 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 since processes are not evaluated like in a functional language but are communicated with (along a channel). This 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 offers along the channel c the session A using channels $c_1, \ldots, c_n$ (linearly) with the corresponding types". We assume $c_1, \ldots, c_n$ and c are all distinct.

Each process offers along a specific channel, and in the linear setting, each channel must be used exactly in one place. Processes cannot rename channels, which means we can treat channel names as unique process identifiers. It is important to note that linearity, or rather the lack of contraction, is crucial to the correctness of the system. Session types describe the behavior of processes and evolve over time as these processes communicate. If we were able to duplicate premises, we would have access to types that processes no longer satisfy, and could easily violate type safety.

Working out the isomorphism further and assigning a session type to each linear proposition gives the following interpretation:

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

The types we care about the most in this paper are $\oplus\{lab_k : A_k\}_{k \in I}$ and $\&\{lab_k : A_k\}_{k \in I}$ which are generalizations of the binary additive disjunction ($\oplus$) and conjunction ($\&$). $\oplus\{lab_k : A_k\}_{k \in I}$ is called an internal choice, since the label is picked by the provider (we always consider the world from the provider's perspective). In the same vein, $\&\{lab_k : A_k\}_{k \in I}$ is an external choice since the choice is made externally by the client. In either case, I is a *finite* non-empty set, the order of labels does not matter, and each label must be unique.

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 means P can contain the channel x as a free variable. We will also use it as a shorthand for substitution, writing P_a for $[a/x]P$ (where $[a/x]P$ is the capture avoiding substitution of a for x in P).

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

An example program will give more intuition about the system. We will look at process-level natural numbers, which will also be our running example in this paper. Note that we will use concrete syntax, but the mapping to abstract syntax presented above should be clear. Also, this example (and almost any interesting one) requires recursive types, which are introduced in the next section. First, we define the interface:

```
type Nat = +{zero : 1, succ : Nat}
```

This states that a process-level natural number is an internal choice of either zero or a successor of another natural. Next, we define two simple processes that implement the interface:

```
z : Nat
‘c <- z =
  ‘c.zero;
  close ‘c
```

```
s : Nat -o Nat
‘c <- s ‘d =
  ‘c.succ;
  ‘c <- ‘d
```

`z` simply sends the label `zero` along the channel `‘c` (which it provides) and terminates, whereas `s` prepends a successor to the number provided along channel `‘d` and then delegates to `‘d`. Here is a slightly more complicated example that uses recursion:

```
double : Nat -o Nat
‘c <- double ‘d =
  case ‘d of
    zero -> wait ‘d; ‘c.zero; close ‘c
    succ -> ‘c.succ; ‘c.succ; ‘c <- double ‘d
```

These are very simple examples, though we hope they offer some insight into the system. We will assign more interesting types to these processes after we introduce intersections and unions.

2.2 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. One thing to note is that in $\oplus$L and &R, we allow unused branches in case expressions. This makes width subtyping easier, as discussed in section 3.3.

2.3 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,

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

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

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

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

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

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

$$\frac{i \in I \quad \Psi, c : A_i \vdash P :: (d : D)}{\Psi, c : \&\{lab_k : A_k\}_{k \in I} \vdash c.lab_i ; P :: (d : D)}\ \&\mathtt{L}$$

Figure 1: Type assignment for process expressions

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 $\mathtt{proc}_c(P)$ 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.

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

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

2.4 Operational Semantics

A process configuration evolves over time when a process takes a step, either by spawning a new process (`cut`), delegation (`id`) or when two matching processes communicate. For example, the processes configuration $\Omega, \mathtt{proc}_c(c.lab_i ; P), \mathtt{proc}_d(\text{case } c \text{ of } \{lab_k \rightarrow Q_k\}_{k \in I})$ can step to $\Omega, \mathtt{proc}_c(P) \otimes \mathtt{proc}_d(Q_i)$

whenever $i \in I$. We express these rules using *substructural operational semantics* [19] which are based on *multiset rewriting* [7]. For example, the rule for $\mathbf{1}$ can be written as:

$$\mathtt{proc}_c(\text{close } c) \otimes \mathtt{proc}_d(\text{wait } c\,;P) \multimap \{\mathtt{proc}_d(P)\}.$$

Note that the rule is written using linear connectives, however, these should not be confused with connectives we used for types. For example, $A \otimes B \otimes C \multimap \{D \otimes E\}$ would mean we could replace the resources A, B, C with D, E. The $\{\ldots\}$ indicates a monad which essentially forces the rules to be interpreted as a multiset rewriting rule. The rest of the rules are given below:

$$\begin{array}{rl}
\mathtt{id} & : \mathtt{proc}_c(c \leftarrow d) \multimap \{c = d\} \\
\mathtt{cut} & : \mathtt{proc}_c(x \leftarrow P_x\,; Q_x) \multimap \{\exists a.\mathtt{proc}_a(P_a) \otimes \mathtt{proc}_c(Q_a)\} \\
\mathtt{one} & : \mathtt{proc}_c(\text{close } c) \otimes \mathtt{proc}_d(\text{wait } c\,;P) \multimap \{\mathtt{proc}_d(P)\} \\
\mathtt{tensor} & : \mathtt{proc}_c(\text{send } c\ (x \leftarrow P_x)\,; Q) \otimes \mathtt{proc}_e(x \leftarrow \text{recv } c\,; R_x) \\
& \quad \multimap \{\exists a.\mathtt{proc}_a(P_a) \otimes \mathtt{proc}_c(Q) \otimes \mathtt{proc}_e(R_a)\} \\
\mathtt{internal} & : \mathtt{proc}_c(c.lab_i\,; P) \otimes \mathtt{proc}_d(\text{case } c \text{ of } \{lab_k \to Q_k\}_{k\in I}) \otimes i \in I \\
& \quad \multimap \{\mathtt{proc}_c(P) \otimes \mathtt{proc}_d(Q_i)\} \\
\mathtt{lolli} & : \mathtt{proc}_c(x \leftarrow \text{recv } c\,; P_x) \otimes \mathtt{proc}_d(\text{send } c\ (x \leftarrow Q_x)\,; R) \\
& \quad \multimap \{\exists a.\mathtt{proc}_c(P_a) \otimes \mathtt{proc}_a(Q_a) \otimes \mathtt{proc}_d(R)\} \\
\mathtt{external} & : \mathtt{proc}_c(\text{case } c \text{ of } \{lab_k \to P_k\}_{k\in I}) \otimes \mathtt{proc}_d(c.lab_i\,; Q) \otimes i \in I \\
& \quad \multimap \{\mathtt{proc}_c(P_i) \otimes \mathtt{proc}_d(Q)\}
\end{array}$$

3 Recursion and Subtyping

Next, we introduce equirecursive types and recursive processes which are central in many applications of session types. We will also mention (the initial version of) the subtyping judgment which is needed to deal with type equivalences induced by equirecursive types.

3.1 Recursive Types

We extend the language of types with variables and a new construct, $\mu t.A_t$, representing recursive types. We require all types at the top level to be closed, and make sure every rule we give preserves this property. Recursive types $\mu t.A$ are identified with their unfolding $[\mu t.A/t]A$ which means there are no explicit term level coercions (`unfold` and `fold`) to go between them. This is the reason they are called equirecursive as opposed to isorecursive where term level coercions would witness the isomorphism. Equirecursive types tend to make type-checking and meta-theory harder, however, they reduce communication and make more sense in a concurrent setting where behavior is more important than term structure.

In the style of [2], we interpret recursive types as finite representations of potentially infinite μ-free types through repeated unfolding. For example, the type $\mu t.\mathbf{1} \multimap t$ stands for $\mathbf{1} \multimap (\mathbf{1} \multimap (\mathbf{1} \multimap (\cdots)))$ and $\mu t.t \otimes t$ represents $(\cdots) \otimes (\cdots)$. For this to make sense, we assume that all types are contractive [20, 12]. Intuitively, this means occurrences of variables must be under a *structural* type.

3.2 Recursive Processes

Term level recursion is achieved by a new form of process expression, rec $p(\bar{c}).P_p$, which is parametrized over channels $\bar{c}$ to allow renaming. The development is fairly standard and is not as important in this paper. More detail can be found in [22]. The only thing to note is that the typing judgment is extended with a new context η to keep track of process variables. The new judgment is written $\Psi \vdash_\eta P :: (c : A)$.

3.3 Subtyping

Gay and Hole [12] 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 structures. Subtyping also doubles as a convenient way of identifying a recursive type and its unfolding (without it, we would need a type equality judgment almost equally complicated but more restrictive). Subtyping is especially important for a refinement system since it is used to propagate refinements and forget them as necessary. We do not go into the details of their 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:

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

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.

4 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:

```
type Nat = +{zero : 1, succ : Nat}

type Pos = +{succ : Nat}
type Even = +{zero : 1, succ : Odd}
type Odd = +{succ : Even}
```

Intuitively, it is easy to see that `Pos`, `Even`, `Odd` are all subtypes of `Nat`. We run into a problem when we try to implement the behavior described by these types, however. Consider the `s` function, for example, which 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.

4.1 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:

```
z : Nat and Even
s : (Nat −o Nat) and (Even −o Odd) and (Odd −o Even)
double : (Nat −o Nat) and (Nat −o Even)
```

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:

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

The standard subtyping rules are given below, where double lines indicate rules should be interpreted coinductively. 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 the subtyping rules are only first attempt.

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

4.2 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 important to point out that we restore a long-lost symmetry for functional languages. The natural left rule we give here for unions (natural since it is dual to the right rule for intersection) has been shown to be problematic in functional languages [3].

One solution limits the left rule to expressions in evaluation position [10]. The straightforward left rule turns out to be already sound here due to our use of the linear sequent calculus.

The usual subtyping rules are given below. These make the right rules derivable so they are not explicitly added to the system.

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

Unions allow us to describe some interesting properties. For example, we can show that every natural is either even or odd:

```
iso : Nat -o (Even or Odd)
‘c <- iso ‘d =
  case ‘d of
    zero -> wait ‘d; ‘c.zero; close ‘c
    succ -> ‘c.succ; ‘e <- iso ‘d; ‘c <- ‘e
```

We have to unfold one level since our system cannot prove $Nat \leq Even \sqcup Odd$. A slightly more involved example is that of bit strings, which uses unions in the type definition and demonstrates “strengthening the induction hypothesis”. As before, we first define the interface:

```
type Bits = +{eps : 1, zero : Bits, one : Bits}
```

Here, `eps` is the empty string, `zero` and `one` append a least significant bit. We can define bit strings in standard form (no leading zeros) as follows:

```
type Empty = +{eps : 1}
type Std = Empty  or StdPos
type StdPos = +{one : Std, zero : StdPos}
```

Then, we can write an increment function that preserves bit strings in standard form:

```
inc : Std -o Std and StdPos -o StdPos and Empty -o StdPos
‘c <- inc ‘d =
  case ‘d of
    eps -> wait ‘d; ‘c.one; ‘c.eps; close ‘c
    zero -> ‘c.one; ‘c <- ‘d
    one -> ‘c.zero; ‘c <- inc ‘d
```

Note that checking this definition just against the type `Std -o Std` will fail, and we need to assign the more general type for the type checking to go through. This is because of the bidirectional nature of our system which essentially requires the type checker to check a fixed point rather than infer the least one. This has proven highly beneficial for providing good error messages even without the presence of intersections and unions [15].

4.3 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.[1]

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).[2] This results in a system reminiscent of [13, 14]. 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, however, this is not trivial to do without breaking type safety.

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

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

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

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

$$\frac{\alpha \Rightarrow \beta, [\mu t.A/t]A}{\alpha \Rightarrow \beta, \mu t.A} \Rightarrow \mu\mathtt{R} \qquad \frac{\alpha, [\mu t.A/t]A \Rightarrow \beta}{\alpha, \mu t.A \Rightarrow \beta} \Rightarrow \mu\mathtt{L}$$

Figure 2: Subtyping with multiple hypothesis and conclusions; coinductively

[1] 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.

[2] We use multisets rather than sets since types have nontrivial equality, so it is not obvious when we should combine them into one. Additionally, we do not want to worry about whether the rest of the context already had a given type before we add it in.

4.4 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 special case of binary external choice: $\&\{\texttt{inl} : A, \texttt{inr} : B\}$. 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 can be checked that these definitions satisfy the typing and subtyping rules for external and internal choices.

5 Algorithmic System

In this section, we prove 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.

5.1 Algorithmic Subtyping

The subtyping judgment we gave is already mostly algorithmic (a necessity of working with coinductive rules), so we only have to tie a couple of loose ends. The first is deciding which rule to pick when multiple are applicable. We apply $\Rightarrow \sqcap\mathtt{R}$, $\Rightarrow \sqcap\mathtt{L}$, $\Rightarrow \sqcup\mathtt{R}$, $\Rightarrow \sqcup\mathtt{L}$, $\Rightarrow \mu\mathtt{R}$, $\Rightarrow \mu\mathtt{L}$ eagerly since these are invertible. At some point, we must hit all structural types due to our contractiveness restriction, at which point we non-deterministically pick a structural rule and continue.

Second, the coinductive nature of typing means we can (and often will) have infinite derivations. We combat this by using a cyclicity check (similar to the one in [12]): 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. A more formal treatment can be found in [20].

5.2 Algorithmic Type-checking

Designing a type checking algorithm is quite simple for the base system where we only have structural types (no recursion 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 helper process to check against. This is solved by adding type annotations in spawning processes so that the new form is $c : A \leftarrow P_c ; Q_c$.

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 $A \leq B$ is free, 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 multiset context multiple conclusion logic just like we did with subtyping. This makes intersection left and union right rules invertible, so they can be applied eagerly.

The former problem is solved by switching to *bidirectional type-checking* where we only check subtyping at the identity rule (delegation). This relies on the subformula property for the sequent calculus, excepting only the cut rule which is annotated. The new judgment is written $\Psi \Vdash_\eta P :: (c : A)$. The full system is given in Appendix A.

5.3 Equivalence to the Declarative System

Next, we show that the algorithmic system is sound and complete with respect to the declarative system (modulo type annotations). Due to space limitations, we can only give very brief proof sketches here. Interested readers are referred to the first author's thesis [1].

Theorem 5.1 (Soundness of Algorithmic Typing)**.** *If $\Psi \Vdash_\eta P :: (c : \alpha)$, then $\bigsqcap \Psi \vdash_{\eta'} [\![P]\!] :: (c : \bigsqcup \alpha)$ where η' is η suitably converted using $\bigsqcap$ and $\bigsqcup$. Here, $\bigsqcap \alpha$ is the intersection of all types in α and $\bigsqcup$ is the union.*

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

Lemma 5.2 (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 5.3 (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 5.2 for SubR and SubL. □

6 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 judgement in the statements and proofs of these results. Type safety for the declarative system follows from its equivalence to the algorithmic system. Again, we are not able to present full proofs for space concerns.

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 6.1 (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 6.2 (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. □

7 Conclusion

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 [21], adding shared channels [5, 18], or considering asynchronous communication [8, 18, 15] 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.

References

[1] Coşku Acay (2016): *Refinements for Session Typed Concurrency*. Undergraduate honors thesis, Carnegie Mellon University.

[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] 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.

[9] 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.

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

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

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

[13] 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.

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

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

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

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

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

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

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

[21] 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.

[22] Bernardo Toninho, Luís Caires & Frank Pfenning (2014): *Corecursion and Non-divergence in Session-Typed Processes*. In: *TGC*, *Lecture Notes in Computer Science* 8902, Springer, pp. 159–175.

A Process Typing in the Algorithmic System

Algorithmic typing rules for processes are given below. Rules for recursive processes are not duplicated since they simply capture sets of types rather than a single type.

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

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

$$\frac{\Psi \Vdash_\eta P :: (c : [\mu t.A/t]A, \alpha)}{\Psi \Vdash_\eta P :: (c : \mu t.A, \alpha)}\ \mu\mathtt{R} \qquad \frac{\Psi, c : (\alpha, [\mu t.A/t]A) \Vdash_\eta P :: (d : \beta)}{\Psi, c : (\alpha, \mu t.A) \Vdash_\eta P :: (d : \beta)}\ \mu\mathtt{L}$$

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

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

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

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

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

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

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

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

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