CARNEGIE MELLON UNIVERSITY

Refinements for Session Typed Concurrency

by

Josh Acay

Undergraduate Thesis

in the

School of Computer Science

Computer Science Department

Advisor: Frank Pfenning

May 2016

"*Oh, wow. That's an intense line of questioning, Snuffles.*"

Summer Smith

"*When life gives you lemons, don't make lemonade. Make life take the lemons back! Get mad! I don't want your damn lemons, what the hell am I supposed to do with these?*"

Cave Johnson

Abstract

Prior work has established the logical connection between linear sequent calculus and session-typed message-passing concurrent computation. The basic system was shown to guarantee strong properties such as session fidelity and deadlock freedom. In this thesis, we extend the basic type system with intersection and union types in order to express multiple behavioral properties of processes in a single type. In the presence of equirecursive types and a natural notion of subtyping, the resulting system turns out to be strong enough to statically prove many useful properties. We present our system and illustrate its expressive power with examples.

Acknowledgements

First and foremost, I would like to thank my advisor Frank Pfenning for his continued support. This thesis would not be possible without his encouragement. In addition, discussions with Max Willsey, Hannah Gommerstadt, and Rokhini Prabhu has been of tremendous help. Finally, I am grateful to Noam Zeilberger, Rowan Davies, and Joshua Dunfield for offering their help whenever needed.

Contents

List of Figures

Chapter 1

Introduction

A concurrent system consists of processes that work together to compute a result. In the message-passing formulation of concurrency, interaction between processes is established by exchanging messages through channels that go between them. Session types are assigned to channels in a type safe setting to prescribe the communication behavior along channels. The processes at each end of a channel must respect this type when using the said channel.

Recently, session-typed message-passing concurrency has been put on the firm foundations of logic by establishing a Curry-Howard correspondence to intuitionistic linear sequent calculus [4, 18, 16]. In this formulation, linear propositions are interpreted as session-types, proofs as processes, and cut elimination as communication. The basic type system described in prior work is enough to guarantee strong properties such as deadlock freedom and session fidelity, and the theory can accommodate many different types of communication such as synchronous or asynchronous (we stick to synchronous communication to keep things simple). On the other hand, many interesting behavioral properties turn out to be inexpressible using the basic type system. This usually leads to having to implement impossible cases (which cannot offer any interesting behavior and generally terminate the program). The type checker cannot verify that these cases are in fact impossible, which is not only annoying but generally leads to errors when those cases turn out to be relevant.

In this thesis, we extend the type system of [18] with intersection and union types in order to specify and statically verify more interesting behavioral properties of processes. Previously, Freeman and Pfenning has shown how intersections can be used as refinements in a conventional functional language [11]. We hope to carry their success into the concurrent setting and expand on it by introducing unions as well.

To do so, we show that the base system extended with intersections, unions, recursive types, and a natural notion of subtyping is type-safe, and the resulting type system admits a type checking algorithm. The former follows from the usual type preservation and progress theorems, which correspond to session fidelity and deadlock freedom in the concurrent setting. We accomplish the latter by giving an algorithmic system and showing its equivalence to the declarative one.

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].

The rest of the thesis is structured as follows. In chapter 2, we give a brief overview of prior work on the correspondence between concurrent computation and linear sequent logic. We introduce the base system and equirecursive types. Chapter 3 extends this system with a natural notion of subtyping. In chapter 4, we add intersection and union types to the system. This necessitates modifying the subtyping relation in order to admit distributivity laws. We present the new subtyping relation and explore its properties. Chapter 5 presents the algorithmic system, and finally, chapter 6 concludes with suggestions for future work.

Chapter 2

From Linear Logic to Session Types

2.1 Linear Propositions as Session Types

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 [4, 18, 16]. 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 by exactly one process. Processes cannot rename channels, which means we can treat channel names as unique process identifiers.

Working out the isomorphism further and assigning a session type to each linear proposition gives the interpretation below. Note that types are interpreted from the perspective of the process providing the type (the provider) rather than the process using it (the client).

A, B, C	$::=$	$\mathbf{1}$	send `end` and terminate
	$\mid$	$A \otimes B$	send channel of type A and continue as B
	$\mid$	$\oplus\{lab_k : A_k\}_{k \in I}$	send lab_i and continue as A_i
	$\mid$	$\tau \wedge B$	send value of type τ and continue as B
	$\mid$	$A \multimap B$	receive channel of type A and continue as B
	$\mid$	$\&\{lab_k : A_k\}_{k \in I}$	receive lab_i and continue as A_i
	$\mid$	$\tau \supset B$	receive value of type τ and continue as B

$\mathbf{1}$ corresponds to processes that offer no interesting behaviour. The types $A \otimes B$ and $A \multimap B$ correspond to sending and receiving channel names respectively. $\oplus\{lab_k : A_k\}_{k \in I}$ is called an internal choice, since the label is picked by the provider. Similarly, $\&\{lab_k : A_k\}_{k \in I}$ is an external choice since the choice is made by the client. In either case, I is a *finite* index set, $lab : I \to \mathbf{Label}$ is an *injective* function into the set of labels, and $A : I \to \mathbf{Type}$ is any function into types. The order of labels does not matter and each label must be unique. Finally, $\tau \wedge B$ and $\tau \supset B$ type processes that send and receive values in some underlying functional language. In this thesis, we will ignore these types and limit our focus to the process calculus. The integration of a functional language is orthogonal and can be found in [22].

2.2 Process Expressions

Within this framework, proof terms correspond to processes. For example, cut, written $c \leftarrow P_c\,;\, Q_c$, denotes a form of process composition where the client spawns off a helper process (P_c) with which it can communicate from then on. The intuition is formalized in the typing rule:

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

The rest of the process expressions are summarized below, with the sending construct followed by the matching receiving construct. Discussion of other typing rules is given in section 2.2.2.

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

Note that cut, `send`, and `recv` bind the spawned, sent, and received channel names, which means these are identified up to alpha conversion. We denote the usual capture avoiding simultaneous substitution of channels $\bar{c}$ for $\bar{x}$ in P by $[\bar{c}/\bar{x}]P$ where $\bar{c}$ and $\bar{x}$ are ordered sequences of equal length.

2.2.1 An Example Process: Process Level Naturals

Let us consider an example program to get more intuition about the system. We will use process level natural numbers, `Nat`, as a running example. Examples are given using concrete syntax, which closely follows the abstract syntax presented above with some syntactic sugar. Briefly, channel names start with a `‘`. Type declarations have the form `type <type name> = <definition>` and process declarations have the form `<offered channel> <- <process name> = <definition>`. Process declarations can refer to the offered channel, and both kinds of declarations can use the declared name recursively.[1] All declarations are considered mutually recursive. We define send $c\,d\,;\,P = \text{send}\;c\;(x \leftarrow x \leftarrow d)\,;\,P$ and $c \leftarrow P = x \leftarrow P\,;\,(c \leftarrow x)$ as syntactic sugar. If a process declaration has a type $A_1 \multimap A_2 \multimap \ldots \multimap A_n \multimap B$, then it can be applied to n channels of the corresponding types, which is translated into a spawn followed by n sends. The full grammar is given in appendix A.

With that out of the way, we can give our first example. First, we define the interface:

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

The interface 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` send the label `succ` and delegates to `‘d`. Here is a slightly more complicated example that uses recursion:

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

[1] Recursive types and processes are introduced formally in section 2.3.

```
succ -> ‘c.succ; ‘c.succ; ‘c <- double ‘d
```

2.2.2 Type Assignment for Processes

The typing rules for other constructs are derived from linear logic by decorating derivations with proof terms. The rules are given in Figure 2.1. Note that we allow unused branches case expressions for $\oplus\mathtt{L}$ and $\&\mathtt{R}$, which makes width subtyping easier (discussed in chapter 3).

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

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

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

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

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

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

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

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

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

FIGURE 2.1: Type assignment to processes

As usual, we identify bound channels up to alpha conversion. Free channels are subject to consistent renaming across a sequent by substitution.

2.3 Recursion

In this section, we introduce equirecursive types and recursive processes which are central in many applications of session types.

2.3.1 Recursive Types

We extend the language of types with variables and a new construct, $\mu t.A_t$, representing recursive types. A recursive type $\mu t.A$ is identified with its unfolding $[\mu t.A/t]A$, which means there are no explicit term level coercions (e.g. `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 [1], 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 $(\cdots) \otimes (\cdots)$. This interpretation, however, breaks down when we have types such as $\mu t.t$ which only unfold to themselves (therefore, no amount of unfolding can remove the μ). To forbid such types, we introduce the standard global syntactic restriction called contractiveness [21, 12] and only consider contractive types from then on.

2.3.1.1 Contractiveness

Intuitively, a recursive type $\mu t.A$ is contractive if all occurrences of t in A are under a *structural* (i.e. not μ) type constructor. For example, $\mu t.\mathbf{1} \multimap t$ and $\mu t.t \otimes t$ are contractive whereas $\mu t.t$ and $\mu t.\mu u.t$ are not.

We formalize contractiveness using the notion of unguarded variables [21]. Unguarded variables of a type A, denoted $\mathrm{UV}(A)$, are defined inductively as follows:

$$\begin{aligned}
\mathrm{UV}(\mathbf{1}) &= \emptyset \\
\mathrm{UV}(A \otimes B) &= \emptyset \\
\mathrm{UV}(\oplus\{lab_k : A_k\}_{k \in I}) &= \emptyset \\
\mathrm{UV}(A \multimap B) &= \emptyset \\
\mathrm{UV}(\&\{lab_k : A_k\}_{k \in I}) &= \emptyset \\
\mathrm{UV}(t) &= \{t\} \\
\mathrm{UV}(\mu t.A) &= \mathrm{UV}(A) \setminus \{t\}
\end{aligned}$$

A type is then said to be contractive if every occurrence of $\mu t.A$ satisfies $t \notin \mathrm{UV}(A)$ as formalized in Figure 2.2.

$$\frac{}{\mathbf{1}\ \texttt{contractive}} \qquad \frac{A\ \texttt{contractive} \quad B\ \texttt{contractive}}{A \otimes B\ \texttt{contractive}} \qquad \frac{A_x\ \texttt{contractive for}\ x \in I}{\oplus\{lab_k : A_k\}_{k \in I}\ \texttt{contractive}}$$

$$\frac{A\ \texttt{contractive} \quad B\ \texttt{contractive}}{A \multimap B\ \texttt{contractive}} \qquad \frac{A_x\ \texttt{contractive for}\ x \in I}{\&\{lab_k : A_k\}_{k \in I}\ \texttt{contractive}}$$

$$\frac{t \notin \mathrm{UV}(A) \quad A\ \texttt{contractive}}{\mu t.A\ \texttt{contractive}}$$

FIGURE 2.2: Contractiveness

Contractiveness ensures that repeated unfolding will terminate with a structural type in a finite number of steps. This is required for soundness of the theory, and comes up often in many inductive arguments we will present. Usually, the property that gets smaller will be the size of a type, which is defined to be the number of unfoldings we need to do before we hit a structural type:

$$\begin{aligned}
\mathrm{size}(\mathbf{1}) &= 0 \\
\mathrm{size}(A \otimes B) &= 0 \\
\mathrm{size}(\oplus\{lab_k : A_k\}_{k \in I}) &= 0 \\
\mathrm{size}(A \multimap B) &= 0 \\
\mathrm{size}(\&\{lab_k : A_k\}_{k \in I}) &= 0 \\
\mathrm{size}(t) &= \bot \\
\mathrm{size}(\mu t.A) &= 1 + \mathrm{size}(A)
\end{aligned}$$

Note that size is well defined for contractive types since we can never hit the variable case. It is of course finite since types are finite.

2.3.1.2 Well-formed Types

Extending the language of types with variables means not every syntactically valid type makes sense. For example, the type $\mu t.u$ is meaningless since u is not bound anywhere. Fortunately, all such types can be eliminated by requiring all types to be closed. That is, the set of free variables of a type should be empty. Just like contractiveness, we will assume all types we consider are closed and will not explicitly restate this assumption. Every operation we need on types preserves this property[2].

2.3.2 Recursive Processes

We introduce a new form of process expression which we write rec $p(\bar{c}).P_p$ which are modeled after the corecursive processes of [23]. Here, p is a process variable that intuitively stands for the whole expression and $\bar{c}$ is an ordered list of channel names that is used to parametrize the expression over channel names. We use the notation $P\ \bar{c}$ for parameter instantiation. Parametrization is useful in case we want to rename the provided or used channels. For instance, we will often want to spawn a copy of the overall expression: rec $p(c).d \leftarrow p\ d\,;\ P_d$ where P_d is some process that consumes d and offers along c. The typing rules limit specialization to recursive processes and process variables.

We also have to extend the typing context to keep track of process variables. Note that we cannot simply add this information to the existing context since that contexts tracks channel names which are different from processes. In addition, the channel context is linear, but there is no reason to limit recursive occurrences of a process to exactly one place. We write the new judgment as $\Psi \vdash_\eta P :: (c : A)$, where η stores the typing context for process variables. As usual, we assume variable names in η are made unique through alpha renaming. Recursive processes are typed using the rules in Figure 2.3. These are the only rules that modify the process variable context, all other rules simply pass it up unchanged.

Note that in the definition of η', $[\bar{y}/\bar{z}]\Psi \vdash p(\bar{y}) :: ([\bar{y}/\bar{z}]c : A)$ is not a typing judgment. Instead, η should be thought of as nothing more than a map from variable names to four tuples containing parameter names, typing context, provided channel name, and provided type. It is necessary to store the context since channels are linear and channel types evolve over time, but the context needs to be the same at every occurrence of p.

[2]We will only unfold a μ which clearly results in a closed type given a closed type. We never take a μ apart. We will break down other types such as $A \otimes B$, but that cannot result in an open type.

$$\frac{\Psi \vdash_{\eta'} [\bar{z}/\bar{y}]P :: (c : A) \quad \eta' = \eta, [\bar{y}/\bar{z}]\Psi \vdash p(\bar{y}) :: ([\bar{y}/\bar{z}]c : A)}{\Psi \vdash_{\eta} (\mathrm{rec}\ p(\bar{y}).P)\ \bar{z} :: (c : A)}\ \mu$$

$$\frac{\Psi \vdash p(\bar{y}) :: (c : A) \in \eta}{[\bar{z}/\bar{y}]\Psi \vdash_{\eta} p\ \bar{z} :: ([\bar{z}/\bar{y}]c : A)}\ \mathtt{var}$$

FIGURE 2.3: Type assignment for recursive processes

2.3.3 Type Equality

We mentioned that we will identify a recursive type $\mu t.A$ with its unfolding $[\mu t.A/t]A$, but we have not yet formally introduced this to the theory. As things currently stand, it is not possible to type any process that requires an unfold. We address that problem by defining an equality relation $A \equiv B$ between types and introduce the conversion rules given in Figure 2.4.

$$\frac{\Psi \vdash_{\eta} P :: (c : A') \quad A' \equiv A}{\Psi \vdash_{\eta} P :: (c : A)} \equiv \mathtt{R} \qquad \frac{\Psi, c : A' \vdash_{\eta} P :: (d : B) \quad A \equiv A'}{\Psi, c : A \vdash_{\eta} P :: (d : B)} \equiv \mathtt{L}$$

FIGURE 2.4: Type conversion

Possibly the more interesting part is our definition of $\equiv$. Intuitively, it is the unfolding rule $\mu t.A \equiv [\mu t.A/t]A$ along with a congruence rule for each structural type constructor. However, we define $\equiv$ *coinductively* since a coinductive definition can safely equate more types [21] and since we are more interested in behaviour than structure as mentioned before. The rules for $\equiv$ are given in Figure 2.5. We use double lines to mean rules should be interpreted coinductively.

$$\frac{}{\mathbf{1} \equiv \mathbf{1}} \equiv \mathbf{1} \qquad \frac{A \equiv A' \quad B \equiv B'}{A \otimes B \equiv A' \otimes B'} \equiv \otimes \qquad \frac{A_x \equiv A'_x \text{ for } x \in I}{\oplus\{lab_k : A_k\}_{k\in I} \equiv \oplus\{lab_k : A'_k\}_{k\in I}} \equiv \oplus$$

$$\frac{A' \equiv A \quad B \equiv B'}{A \multimap B \equiv A' \multimap B'} \equiv \multimap \qquad \frac{A_x \equiv A'_x \text{ for } x \in I}{\&\{lab_k : A_k\}_{k\in I} \equiv \&\{lab_k : A'_k\}_{k\in I}} \equiv \&$$

$$\frac{A \equiv [\mu t.B/t]B}{A \equiv \mu t.B} \equiv \mu\mathtt{R} \qquad \frac{[\mu t.A/t]A \equiv B}{\mu t.A \equiv B} \equiv \mu\mathtt{L}$$

FIGURE 2.5: Equality of types

We expect type equality to be an equivalence relation between types (i.e. it should be reflexive, symmetric, and transitive). In a coinductive setting, however, adding symmetry

and/or transitivity explicitly will make all types equal! We have thus carefully constructed the rules so that these properties are admissible as proven next.

Theorem 2.1. $\equiv$ *is an equivalence:*

- $A \equiv A$ *for all types* A.
- $A \equiv B$ *implies* $B \equiv A$ *for all types* A, B.
- $A \equiv B$ *and* $B \equiv C$ *implies* $A \equiv C$ *for all types* A, B, C.

Proof. Reflexivity is by coinduction on A. Symmetry follow from a simple coinduction on the derivation of type equality using the symmetric rules for $\equiv \mu\mathtt{R}$ and $\equiv \mu\mathtt{L}$. For transitivity, we use a lexicographic combination of coinduction on the two equality derivations and induction on size (B).

The details are omitted since the proofs are quite standard (except for the use of coinduction, which does not change the structure of the proof). In addition, we will soon replace equality with subtyping in chapter 3. □

2.4 Process Configurations

So far, we have only considered processes in isolation. In a concurrent setting, we need to be able to talk about multiple processes and the interactions between them. In this section, we introduce process configurations, which are simply sets 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 that all labels in a configuration are distinct. That is, a process configuration $\{\mathtt{proc}_{c_1}(P_1), \ldots, \mathtt{proc}_{c_n}(P_n)\}$ is valid if and only if $c_1, \ldots, c_n$ are all distinct. Note that we do not require channels that occur within P_i to be distinct, this is handled by the typing judgment given next.

Definition 2.2 (Process Configuration)**.** $\Omega = \{\mathtt{proc}_{c_1}(P_1), \ldots, \mathtt{proc}_{c_n}(P_n)\}$ is called a process configuration if $c_1, \ldots, c_n$ are all 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 (including root nodes which have "dangling" edges disconnected on one side) and any number of incoming edges that correspond to the

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

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

FIGURE 2.6: Configuration typing

channels the process uses. This observation suggests the typing rules in Figure 2.6, which mimic the structure of a multi-way tree, for a process configuration.

This definition is well-founded since the size of the configuration gets strictly smaller. The rules only expose the types of the roots since this is the only information we need when typing the next level. At the top level, we will usually start with one process with type $\mathbf{1}$, which will spawn off providers as needed using `cut`. Since we do not care about the specific type at the top level, we say a process configuration Ω is well typed if $\models \Omega :: \Psi$ for some Ψ. Finally, note that the rules do not allow cyclic uses of channel names, and that the left of the turnstile is empty since configurations must be closed.

Definition 2.3 (Domain of a Configuration)**.** We define

$$\mathrm{dom}\,\{\mathtt{proc}_{c_1}(P_1), \ldots, \mathtt{proc}_{c_n}(P_n)\} = \{c_1, \ldots, c_n\}.$$

Type equality interacts with configuration typing as expected:

Lemma 2.4. *If* $\models \Omega :: \Psi$ *and* $\Psi \equiv \Psi'$ *then* $\models \Omega :: \Psi'$.

Proof. Case for multiple channels follows immediately from the induction hypotheses. Single channel case is by inversion and $\equiv \mathtt{R}$. □

The following inversion lemma will come in handy:

Lemma 2.5 (Inversion of Configuration Typing)**.** *If* $\models \Omega :: \Psi$ *and* $\mathtt{proc}_c(P) \in \Omega$*, then there exist* Ω_1, Ω_2 *such that* $\Omega = \Omega_1, \Omega_2, \mathtt{proc}_c(P)$ *and* $\models \Omega_2, \mathtt{proc}_c(P) :: (c : A)$ *for some* A*. In addition, for any* Ω_2' *and* P' *such that* $\models \Omega_2', \mathtt{proc}_c(P') :: (c : A)$*,* $\models \Omega_1, \Omega_2', \mathtt{proc}_c(P') :: \Psi$.

Proof. By a straightforward induction on the typing derivation. □

2.5 Dynamic Semantics

2.5.1 Reduction

We express reduction rules using *substructural operational semantics* [20] which are based on *multiset rewriting* [5]. 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. Instead, a rule of the form $A_1 \otimes \ldots \otimes A_n \multimap \{B_1 \otimes \ldots \otimes B_m\}$ means we can replace the resources $A_1, \ldots, A_n$ with $B_1, \ldots, B_m$. The curly braces $\{\ldots\}$ indicates a monad which essentially forces the rules to be interpreted as a multiset rewriting rule. $\{\exists x.F\}$ generates a fresh a and proceeds with $[a/x]F$, while $c = d$ performs a global identification of c and d in the configuration. The rest of the rules are given in Figure 2.7.

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

FIGURE 2.7: Reduction rules for process configurations

We say that Ω reduces to or steps to Ω' if one application of the above rules transforms Ω into Ω', and write $\Omega \longrightarrow \Omega'$. We denote the reflexive transitive closure of $\longrightarrow$ by $\longrightarrow^*$.

An important observation about reductions in typed configurations is that they are constrained to one subtree, where the tree structure is implicit in the typing judgment as discussed in section 2.4. This observation leads to the following framing rule.

Definition 2.6 (Root of Reduction)**.** We say that channel c is the root of $\Omega \longrightarrow \Omega'$ if $\texttt{proc}_c(P) \in \Omega$ is rewritten by the reduction, and either it is the only process to the left of $\multimap$ (rules $\texttt{id}, \texttt{cut}, \texttt{rec}$), or it is the client.

Lemma 2.7 (Framing)**.** *If* $\models \Omega :: \Psi$ *and* $\Omega \longrightarrow \Omega'$ *then there exist* $\Omega_1, \Omega_2, \Omega_2'$ *such that* $\Omega = (\Omega_1, \Omega_2)$*,* $\Omega' = (\Omega_1, \Omega_2')$*,* $\Omega_2 \longrightarrow \Omega_2'$*, and* $\models \Omega_2 :: (c : A)$ *where* c *is the root of* $\Omega \longrightarrow \Omega'$*. In addition, if* $\models \Omega_2' :: (c : A)$*, then* $\models \Omega' :: \Psi$*.*

Proof. Assume $\models \Omega :: \Psi$ and $\Omega \longrightarrow \Omega'$ with root c. We proceed by induction on $\models \Omega :: \Psi$.

- For the single channel case, assume $\Psi = d : A$. If $d = c$, then we pick $\Omega_1 = \emptyset$, $\Omega_2 = \Omega$, and $\Omega_2' = \Omega'$. The result follows immediately. Otherwise, we know $\texttt{proc}_c(P) \in \Omega$ was not part of the reduction, so we can apply the induction hypothesis. We add $\texttt{proc}_c(P)$ to the Ω_1 we get from the induction hypothesis.
- For the multiple channel case, assume $\Psi = d_1 : A_1, \ldots, d_n : A_n$, $\Omega = \Omega^1, \ldots, \Omega^n$, and $\models \Omega^i :: (d_i : A_i)$. We know the reduction must work on only one of the Ω_i, so we apply the induction hypothesis on that portion of the context. We add the rest to Ω_1.

□

2.5.2 Progress

In a conventional functional language, the progress theorem states that a well-typed expression either is a value or it takes a reduction step. We do not have values in a process calculus, but there is a corresponding notion we call being poised. Intuitively, a process is poised if it is waiting on its client. We introduce a new judgment $\texttt{proc}_c(P)\ \texttt{poised}$ capturing this notion and define it in Figure 2.8.

$$\frac{}{\texttt{proc}_c(\text{close } c)\ \texttt{poised}} \qquad \frac{}{\texttt{proc}_c(\text{send } c\ (d \leftarrow P_d)\,;\, Q)\ \texttt{poised}}$$

$$\frac{}{\texttt{proc}_c(x \leftarrow \text{recv } c\,;\, Q_x)\ \texttt{poised}} \qquad \frac{}{\texttt{proc}_c(c.lab_i\,;\, P)\ \texttt{poised}}$$

$$\frac{}{\texttt{proc}_c(\text{case } c \text{ of } \{lab_k \rightarrow Q_k\}_{k \in I})\ \texttt{poised}}$$

FIGURE 2.8: Poised processes

We say that a process configuration Ω is poised if every process in Ω is poised. We will need the following inversion lemma about well-typed poised processes to handle type equality:

Lemma 2.8 (Inversion of Process Typing)**.** *If* $\Psi \vdash_\emptyset P :: (c : A)$ *and* $\texttt{proc}_c(P)$ `poised`*, then:*

- *If* $A \equiv \mathbf{1}$ *then* $P = \text{close } c$.
- *If* $A \equiv A_1 \otimes A_2$ *then* $P = \text{send } c\ (d \leftarrow Q_d)\,;\ P'$.
- *If* $A \equiv \oplus\{lab_k : A_k\}_{k \in I}$ *then* $P = c.lab_x\,;\ P'$ *where* $x \in I$.
- *If* $A \equiv A_1 \multimap A_2$ *then* $P = d \leftarrow \text{recv } c\,;\ P'$.
- *If* $A \equiv \&\{lab_k : A_k\}_{k \in I}$ *then* $P = \text{case } c \text{ of } \{lab_k \to P'_k\}_{k \in J}$ *where* $I \subseteq J$.

Proof. The proof is by induction on the derivation of $\Psi \vdash_\emptyset P :: (c : A)$ where Ψ and A are free.

Case $\texttt{id}, \texttt{cut}, \mathbf{1}\texttt{L}, \otimes\texttt{L}, \oplus\texttt{L}, \multimap\texttt{L}, \&\texttt{L}, \mu$: Impossible since P is poised.

Case $\mathbf{1}\texttt{R}, \otimes\texttt{R}, \oplus\texttt{R}, \multimap\texttt{R}, \&\texttt{R}$: If the rule matches the expected type (e.g. $A \equiv 1$ and the rule is $\mathbf{1}\texttt{R}$), then P has the expected form and we are done. Otherwise, we use inversion on the type equality judgment to show that the case is impossible.

Case $\equiv \texttt{L}$: Follows immediately from the induction hypothesis.

Case $\equiv \texttt{R}$: Follows from the induction hypothesis and the fact that $\equiv$ is transitive (theorem 2.1).

□

We are now ready to state the progress theorem.

Theorem 2.9 (Progress)**.** *If* $\models \Omega :: \Psi$ *then either*

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

Proof. The proof is by induction on configuration typing followed by a nested induction on the typing derivation in the single channel case.

- The case for multiple channels is simpler, so we will do that first. Assume $\Omega = \Omega_1, \ldots, \Omega_n$. By the induction hypothesis, either Ω_i is poised or it takes a step (where $i \in \{1, \ldots, n\}$). If any Ω_i takes a step then Ω takes a step. Otherwise, all Ω_i are poised, so Ω is poised.

- For the single channel case, we know $\Omega = \Omega', \texttt{proc}_c(P)$. By inversion, $\Psi_0 \vdash_\emptyset P :: (c : A)$ and $\models \Omega' :: \Psi_0$. By the induction hypothesis, either Ω' takes a step, in which case Ω takes a step and we are done, or Ω' is poised. Assume Ω' is poised.

  Define $\mathcal{P}(\Psi' \vdash_\emptyset P :: (c : A')) :=$ if $\models \Omega' :: \Psi'$ then either $\texttt{proc}_c(P)$ is poised or Ω can take a step. We proceed by induction on the typing derivation.

  Case $\texttt{id}$: P has the form $c \leftarrow d$ and Ω steps by $\texttt{id}$.

  Case $\texttt{cut}$: P has the form $d \leftarrow Q_d \,;\, P'_d$ and Ω steps by $\texttt{cut}$.

  Case $\mathbf{1}\texttt{R}, \otimes\texttt{R}, \oplus\texttt{R}, \multimap \texttt{R}, \&\texttt{R}$: $\texttt{proc}_c(P)$ is poised.

  Case $\mathbf{1}\texttt{L}, \otimes\texttt{L}, \oplus\texttt{L}, \multimap \texttt{L}, \&\texttt{L}$: The proofs for all these cases follow the same structure, so we will only present $\mathbf{1}\texttt{L}$. We know $P = \text{wait } d \,;\, P'$ for some P' and $d : \mathbf{1} \in \Psi'$. By inversion on $\models \Omega' :: \Psi'$, we get $\Psi'' \vdash_\emptyset Q :: (d : \mathbf{1})$ and $\texttt{proc}_d(Q) \in \Omega'$ for some Ψ'' and Q. $\texttt{proc}_d(Q)$ is poised since Ω' is poised by the outer induction hypothesis, so lemma 2.8 gives $Q = \text{close } d$. Thus, Ω steps by $\texttt{one}$.

  Case μ : P has the form $(\text{rec } t(\bar{y}).P')\ \bar{z}$ and Ω steps by $\texttt{rec}$.

  Case $\equiv \texttt{R}$: Follows immediately by the induction hypothesis.

  Case $\equiv \texttt{L}$: $\Psi' = \Psi'', d : D$ and $\Psi'', d : D' \vdash_\emptyset P :: (c : A')$ for some D' where $D \equiv D'$. We then know $\Psi' \equiv \Psi'', d : D'$, thus, the result follows from lemma 2.4 and the induction hypothesis.

  Finally, we get either $\texttt{proc}_c(P)\ \texttt{poised}$ or Ω steps from $\mathcal{P}(\Psi_0 \vdash_\emptyset P :: (c : A))$ and $\models \Omega' :: \Psi_0$. In the former case, Ω is poised since Ω' is poised from before, and in the latter case we are immediately done.

□

2.5.3 Type Preservation

Preservation is a bit more tedious to prove.

Theorem 2.10 (Preservation)**.** *If* $\models \Omega :: \Psi$ *and* $\Omega \longrightarrow \Omega'$ *then* $\models \Omega' :: \Psi$.

Proof. By lemma 2.7, it suffices to consider the subtree which types the root of reduction. So, assume $\Omega_1 \longrightarrow \Omega_2$ and $\models \Omega_1 :: (c : A)$ where c is the root of $\Omega_1 \longrightarrow \Omega_2$. We need to show $\models \Omega_2 :: (c : A)$.

The proof is by simultaneous case analysis on $\Omega_1 \longrightarrow \Omega_2$ and induction on the typing derivation of the root process, followed by induction on the typing derivation of the provider in cases where there is communication. We need induction rather than simple

inversion due to $\equiv \mathtt{R}$ and $\equiv \mathtt{L}$, which change types on the right and the left (respectively) without exposing the structure of the process.

By inversion, $\Omega_1 = (\Omega_1^c, \mathtt{proc}_c(P))$, $\Psi \vdash_\emptyset P :: (c : A)$, and $\models \Omega_1^c :: \Psi$. Define $\mathcal{P}(\Psi_c \vdash_\emptyset P :: (c : A')) :=$ if $\models \Omega_1^c :: \Psi_c$ then $\models \Omega_2 :: (c : A')$. We proceed by induction.

- $\mathcal{P}\left(\dfrac{}{d : A' \vdash_\emptyset c \leftarrow d :: (c : A')}\ \mathtt{id}\right)$:
  Then, $\Omega_2 = [c/d]\Omega_1^c$. Note that $c \notin \mathrm{dom}\,(\Omega_1^c)$ (since well-formed contexts do not have duplicate channel labels), so $\models \Omega_1^c :: (d : A')$ implies $\models \Omega_2 :: (c : A')$ by substitution.
- $\mathcal{P}\left(\dfrac{\mathcal{D} : \Psi_c \vdash_\emptyset P'_d :: (d : D) \quad \mathcal{E} : \Psi'_c, d : D \vdash_\emptyset c :: (Q_d : A')}{\Psi_c, \Psi'_c \vdash_\emptyset d \leftarrow P'_d\,;\, Q_d :: (c : A')}\ \mathtt{cut}\right)$:
  Then, $\Omega_2 = \Omega_1^c, \mathtt{proc}_a(P'_a), \mathtt{proc}_c(Q_a)$ where a is fresh. By inversion on $\models \Omega_1^c :: (\Psi_c, \Psi'_c)$, there are Ω_1^1, Ω_1^2 such that $\Omega_1^c = (\Omega_1^1, \Omega_1^2)$, $\models \Omega_1^1 :: \Psi_c$ and $\models \Omega_1^2 :: \Psi'_c$. Applying $\mathtt{config}_1$ on $\mathcal{D}$ and $\models \Omega_1^1 :: \Psi_c$ gives $\models \Omega_1^1, \mathtt{proc}_a(P'_a) :: (a : D)$ (through suitable substitution). Then, $\models \Omega_1^2, \Omega_1^1, \mathtt{proc}_a(P'_a) :: (\Psi'_c, a : D)$ by $\mathtt{config}_n$. Finally, $\mathcal{E}$ and $\mathtt{config}_1$ implies $\models \Omega_2 :: (c : A')$.
- $\mathbf{1}\mathtt{R}, \otimes\mathtt{R}, \oplus\mathtt{R}, \multimap \mathtt{R}, \&\mathtt{R}$: Impossible since $\mathtt{proc}_c(P)$ is the client.
- $\mathbf{1}\mathtt{L}, \otimes\mathtt{L}, \oplus\mathtt{L}, \multimap \mathtt{L}, \&\mathtt{L}$: In all cases, $\Psi_c = \Psi'_c, d : D$ for some D. Inversion on $\models \Omega_1^c :: \Psi_c$ gives $\Omega_1^c = \Omega_1^{c'}, \Omega_1^d, \mathtt{proc}_d(Q)$ such that $\Psi' \vdash_\emptyset Q :: (d : D)$, $\models \Omega_1^d :: \Psi'$, and $\models \Omega_1^{c'} :: \Psi'_c$.

  Define $\mathcal{Q}(\Psi_d \vdash_\emptyset Q :: (d : D')) :=$ if $\models \Omega_1^d :: \Psi_d$ and $D' \equiv D$ then $\models \Omega_2 :: (c : A')$. Note that $\mathcal{Q}(\Psi' \vdash_\emptyset Q :: (d : D))$ will give the final result once we prove $\mathcal{Q}$, which we do by induction.

  - $\mathcal{Q}\left(\dfrac{}{\emptyset \vdash_\emptyset \text{close } d :: (d : \mathbf{1})}\ \mathbf{1}\mathtt{R}\right)$, $\mathcal{P}\left(\dfrac{\mathcal{D} : \Psi'_c \vdash_\emptyset P' :: (c : A')}{\Psi'_c, d : \mathbf{1} \vdash_\emptyset \text{wait } d\,;\, P' :: (c : A')}\ \mathbf{1}\mathtt{L}\right)$:
    Then, $\Omega_2 = \Omega_1^{c'}, \Omega_1^d, \mathtt{proc}_c(P') = \Omega_1^{c'}, \mathtt{proc}_c(P')$ (last equality is by inversion on $\models \Omega_1^d :: \emptyset$). $\mathtt{config}_1$ on $\models \Omega_1^{c'} :: \Psi'_c$ and $\mathcal{D}$ gives the desired result.
  - $\mathcal{Q}\left(\dfrac{\mathcal{D} : \Psi_d \vdash_\emptyset R_e :: (e : D'_1) \quad \mathcal{E} : \Psi'_d \vdash_\emptyset Q' :: (d : D'_2)}{\Psi_d, \Psi'_d \vdash_\emptyset \text{send } d\ (e \leftarrow R_e)\,;\, Q' :: (d : D'_1 \otimes D'_2)}\ \otimes\mathtt{R}\right)$,
    $\mathcal{P}\left(\dfrac{\mathcal{F} : \Psi'_c, x : D_1, d : D_2 \vdash_\emptyset P'_x :: (c : A')}{\Psi'_c, d : D_1 \otimes D_2 \vdash_\emptyset x \leftarrow \text{recv } d\,;\, P'_x :: (c : A')}\ \otimes\mathtt{L}\right)$:
    Then $\Omega_2 = \Omega_1^{c'}, \Omega_1^d, \mathtt{proc}_a(R_a), \mathtt{proc}_d(Q'), \mathtt{proc}_c(P'_a)$ where a is fresh. Inversion on $D'_1 \otimes D'_2 \equiv D_1 \otimes D_2$ gives $D'_1 \equiv D_1$ and $D'_2 \equiv D_2$. There are Ω_1^1, Ω_1^2 such that $\models \Omega_1^1 :: \Psi_d$ and $\models \Omega_1^2 :: \Psi'_d$ by inversion on $\models \Omega_1^d :: (\Psi_d, \Psi'_d)$.
    $\mathtt{config}_1$ on $\models \Omega_1^1 :: \Psi_d$ and $\mathcal{D}$ with $\equiv \mathtt{R}$ gives $\models \Omega_1^1, \mathtt{proc}_a(R_a) :: (a : D_1)$. Similarly, $\mathtt{config}_1$ on $\models \Omega_1^2 :: \Psi'_d$ and $\mathcal{E}$ with $\equiv \mathtt{R}$ gives $\models \Omega_1^2, \mathtt{proc}_d(Q') :: (d :$

D_2). Finally, $\texttt{config}_1$ using the previous two derivations, $\models \Omega_1^{c'} :: \Psi'_c$, and $\mathcal{F}$ with gives the desired result.

- $\oplus\texttt{R}, \multimap \texttt{R}, \&\texttt{R}$: Similar to above.
- $\mathcal{Q}\left(\dfrac{\mathcal{D} : \Psi_d \vdash_\emptyset Q :: (d : D'') \quad \mathcal{E} : D'' \equiv D'}{\Psi_d \vdash_\emptyset Q :: (d : D')} \equiv \texttt{R}\right)$:
  $D'' \equiv D$ by transitivity of $\equiv$ (theorem 2.1), so we can immediately apply the induction hypothesis on $\mathcal{D}$.
- $\mathcal{Q}\left(\dfrac{\mathcal{D} : \Psi_d, e : E' \vdash_\emptyset Q :: (d : D') \quad \mathcal{E} : E \equiv E'}{\Psi_d, e : E \vdash_\emptyset Q :: (d : D')} \equiv \texttt{R}\right)$:
  $\Psi_d, e : E \equiv \Psi_d, e : E'$ using $\mathcal{E}$, so $\models \Omega_1^d :: (\Psi_d, e : E')$ by lemma 2.4. Thus, we can apply the induction hypothesis on $\mathcal{D}$, which gives the desired result.
- $\texttt{id}, \texttt{cut}, \mathbf{1}\texttt{L}, \otimes\texttt{L}, \oplus\texttt{L}, \multimap \texttt{L}, \&\texttt{R}, \mu$: Not applicable since we know the form of Q by the outer induction and inversion over $\Omega_1 \longrightarrow \Omega_2$.

- $\mathcal{P}\left(\dfrac{\mathcal{D} : \Psi_c \vdash_\emptyset P :: (c : A'')}{\Psi_c \vdash_\emptyset P :: (c : A')}\ \mu\right)$:
  Follows from a suitable substitution lemma for process variables. We omit the details since it is quite standard.
- $\mathcal{P}\left(\dfrac{\mathcal{D} : \Psi_c \vdash_\emptyset P :: (c : A'') \quad \mathcal{E} : A'' \equiv A'}{\Psi_c \vdash_\emptyset P :: (c : A')} \equiv \texttt{R}\right)$:
  Induction hypothesis on $\mathcal{D}$ gives $\models \Omega_2 :: (c : A'')$. The result follows from lemma 2.4 using $\mathcal{E}$.
- $\mathcal{P}\left(\dfrac{\mathcal{D} : \Psi_c, d : D' \vdash_\emptyset P :: (c : A') \quad \mathcal{E} : D \equiv D'}{\Psi_c, d : D \vdash_\emptyset P :: (c : A')} \equiv \texttt{L}\right)$:
  $D \equiv D'$ implies $\Psi, d : D \equiv \Psi, d : D'$ by definition. Lemma 2.4 gives $\models \Omega_1^c :: (\Psi, d : D')$, which means we can apply the induction hypothesis on $\mathcal{D}$ to get the desired result.

Left out cases are impossible since $\Omega_1 \longrightarrow \Omega_2$ and the structure of the root process must match up. Finally, $\mathcal{P}(\Psi \vdash_\emptyset P :: (c : A))$ gives the desired result. □

Chapter 3

Subtyping

Subtyping is a binary relation between types which captures the notion of being more specific or carrying more information. We say that a type A is a subtype of B if a process offering the type A can safely be used in any context where we expect a process offering the type B. We write $A \leq B$ to mean A is a subtype of B. According to this interpretation, for example, natural numbers would be a subtype of reals since every natural number is also a real number. Depending on the type system and the precise definitions of these types, this may or may not turn out to be the case. Of course, we are interested in processes not algebra, so this example is for intuition only.

The notion of session subtyping we use is closely modeled on that of Gay and Hole [12], whose system has width and depth subtyping for n-ary choices, which are natural for record-like structures. Subtyping also doubles as a convenient way of identifying a recursive type and its unfolding, thus, subsuming and generalizing the type equality we introduced before. In a refinement system, subtyping is especially important since it is used to propagate refinements and erase them as necessary (when interfacing legacy code for example). This last point is deferred to section 4.3 since we are still introducing the base system.

As usual, we introduce subtyping to term typing using what are called subsumption rules, which are presented in Figure 3.1. The right rule says that if a process provides a type A', it can also be seen as providing a (less specific) supertype A. Dually, the left rule says that if a process can properly handle a type A', then it does not hurt to make the type more specific.

With that in mind, we can now talk about the specifics of the actual subtyping relation.

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

FIGURE 3.1: Subsumption rules

3.1 Choice

The first form of subtyping we have is width subtyping for internal and external choices, which is similar to width subtyping for record calculi. The idea of width subtyping is simple: whenever a process offers a set of options, we can use it in a context that requires a subset of those options since we can safely ignore the ones we do not care about (external choice), and similarly, whenever we are prepared to handle a set branches, we are prepared to handle a subset of those branches (internal choice). This suggest the following subtyping rules for external and internal choice respectively: $\&\{lab_k : A_k\}_{k\in I} \leq \&\{lab_k : A_k\}_{k\in J}$ whenever $J \subseteq I$, and $\oplus\{lab_k : A_k\}_{k\in I} \leq \oplus\{lab_k : A_k\}_{k\in J}$ whenever $I \subseteq J$.

The next natural step is to carry this (and any other forms of subtyping we may have) out recursively, that is, to allow the A_k to be replaced by subtypes. This is sometimes called depth subtyping and yields the rules in Figure 3.2. The rules are coinductive due to the same reasons the rules for $\equiv$ are coinductive (refer to section 2.3.3).

$$\frac{I \subseteq J \quad A_k \leq A'_k \text{ for } k \in I}{\oplus\{lab_k : A_k\}_{k\in I} \leq \oplus\{lab_k : A'_k\}_{k\in J}} \leq \oplus \qquad \frac{J \subseteq I \quad A_k \leq A'_k \text{ for } k \in J}{\&\{lab_k : A_k\}_{k\in I} \leq \&\{lab_k : A'_k\}_{k\in J}} \leq \&$$

FIGURE 3.2: Subtyping internal and external choices

Internal and external choices are the main tools we use to define data types, which makes width and depth subtyping is especially important for our refinement system since they allow removing branches altogether and constraining the remaining ones. Combined with recursive subtyping, this will let us define interesting behavioral properties.

3.2 Recursive Types

Subtyping would not be very useful if we did not push it through recursive types. A basic notion of recursive subtyping turns out to be not that much more complicated than recursive type equality.[1] Just as we considered two recursive types equivalent whenever their infinite unfoldings were equivalent (refer to section 2.3.1), we will consider a recursive

[1] Subtyping in the presence of intersections and unions turns out to be more complicated, but that comes later in section 4.3.

type to be a subtype of another if their infinite unfoldings have this property. The rules are given in Figure 3.3.

$$\frac{A \leq [\mu t.B/t]B}{A \leq \mu t.B} \leq \mu\mathtt{R} \qquad \frac{[\mu t.A/t]A \leq B}{\mu t.A \leq B} \leq \mu\mathtt{L}$$

FIGURE 3.3: Subtyping recursive types

Another advantage of handling subtyping properly is that it subsumes type equivalence since we can define $A \equiv B$ if and only if $A \leq B$ and $B \leq A$. This makes type equivalence a derived notion rather than a primitive one, which simplifies the theory. In fact, type equality will not ever come up in the theory since the only reason for introducing it was to identify a recursive type and its unfolding, which will now be handled by subtyping.

3.3 Structural Types and the Complete System

Subtyping is not directly related to the other constructs in the type system, so the rest of the rules are for congruence only. The resulting system is presented in Figure 3.4. Note that $\multimap$ is contravariant on the left as is usual for function-like constructs.

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

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

$$\frac{A \leq [\mu t.B/t]B}{A \leq \mu t.B} \leq \mu\mathtt{R} \qquad \frac{[\mu t.A/t]A \leq B}{\mu t.A \leq B} \leq \mu\mathtt{L}$$

FIGURE 3.4: Subtyping for the base system

Subtyping extends to contexts in the obvious way: $\Psi \leq \Psi'$ if and only if $\Psi = (c_1 : A_1, \ldots, c_n : A_n)$, $\Psi' = (c_1 : A'_1, \ldots, c_n : A'_n)$, and $A_i \leq A'_i$ for $i \in \{1, \ldots, n\}$.

3.4 Metatheory

Just like we did for type equivalence, we expect the subtyping relation to satisfy certain properties. Mainly, it should be a preorder, that is, it should satisfy reflexivity and transitivity. The following theorem shows these are admissible:

Theorem 3.1. *$\leq$ is a preorder:*

- *$A \leq A$ for all types A.*
- *$A \leq B$ and $B \leq C$ implies $A \leq C$ for all types A, B, C.*

Proof. Follows from a simple coinduction for reflexivity. For transitivity, we use a lexicographic combination of coinduction on the two subtyping derivations and induction on size (B). We omit details here since the proof is standard and since we will be switching to a different subtyping relation in section 4.3. □

3.4.1 Type Safety

We did not add new forms of processes, so reduction and the notion of being poised is the same as in section 2.5. We have, however, replaced type equality with a more general notion of subtyping, so we need to revisit the progress and preservation theorems.

The proofs of progress and preservation theorems are slight modifications of the ones we presented in section 2.5: we simply replace every occurrence of $\equiv$ with $\leq$. It should be straightforward to modify the process inversion lemma (lemma 2.8) which is used in the proof of the progress theorem, and the two cases related to $\equiv$, so we omit details here.

Chapter 4

Intersection and Union Types as Refinements

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}
```

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 2.2 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 for intersections are given below. It should be noted that $\sqcap\mathtt{L}_1$ and $\sqcap\mathtt{L}_2$ become redundant with the addition of $\leq \sqcap\mathtt{L}_1$ and $\leq \sqcap\mathtt{L}_2$ since they are derivable by an application of subsumption on the left using these rules.

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

Since we are extending the language of types, we need to revisit contractiveness. Since they do not have a corresponding expression level construct, intersections are not considered structural types. Thus, they propagate unguarded variables:

$$\mathrm{UV}\,(A \sqcap B) = \mathrm{UV}\,(A) \cup \mathrm{UV}\,(B)$$
$$\mathrm{size}\,(A \sqcap B) = 1 + \mathrm{size}\,(A) + \mathrm{size}\,(B)$$

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 [2]. 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. Dual to intersections, the right typing rules become redundant with the addition of the subtyping rules.

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

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 more involved example is given in appendix B.

Definitions of unguarded variables and size are extended similarly:

$$\mathrm{UV}\,(A \sqcup B) = \mathrm{UV}\,(A) \cup \mathrm{UV}\,(B)$$
$$\mathrm{size}\,(A \sqcup B) = 1 + \mathrm{size}\,(A) + \mathrm{size}\,(B)$$

4.3 Subtyping Revisited

Every refinement system requires a notion of subtyping to be practical since subtyping is needed in order to implicitly propagate refinements. For example, if we have a process providing `Pos`, we should be able to use it in a context that requires `Nat` since `Pos` is a more specific type. Otherwise, we would require explicit coercions to erase extra information which can easily become cumbersome, especially when we have multiple refinements on a type and we need a specific subset. However, addition of non-structural types such as intersection and union complicates subtyping since these types do not depend on or reveal the structure of the processes they describe. More specifically, property types raise questions about the soundness of completeness of the subtyping relation.

Subtyping is said to be sound if whenever $A \leq B$, using processes of type A in contexts expecting a process of type B does not break type safety. This is usually requires (1) all terms of types A and B to have the same structure, and (2) the set of possible behaviors of terms with type A to be a subset of the set of possible behaviors of terms with type B. Some systems, mainly the ones that use coercive subtyping [17], may not necessarily require condition (1). We will in this system though, since we do not want term level constructs (whether explicit or implicit) that witness convertibility. Subtyping is complete if every time it is safe to use A for B, we have $A \leq B$.

Soundness of subtyping is necessary for type safety, so we have no choice but to make sure this is the case. On the other hand, type safety will hold even in the presence of incompleteness. In fact, most practical systems give up on completeness since it usually turns out to be very hard to design a simple and complete subtyping relation. This will be the case for our system, though, we do try to find a middle ground between simplicity and completeness in order to recover some rules we believe are important in practice.

4.3.1 Distributivity Laws

The subtyping relation we give in chapter 3 and in the previous sections is not complete with respect to, say, the ideal semantics of types [24, 7]. This is because intersections and unions admit many distributivity-like rules over structural types and over each other. For

example, it is not hard to see that $(A_1 \otimes A_2) \sqcap (B_1 \otimes B_2) \leq (A_1 \sqcap B_1) \otimes (A_2 \sqcap B_2)$ would be sound using a propositional reading: if a process sends out a channel that satisfies A_1 then acts as B_1, *and* the sent channel also satisfies A_2 in addition to the result satisfying B_2, then the channel satisfies both A_1 and A_2 and the result satisfies both B_1 and B_2. However, this judgment is not admissible in the given system: the only applicable rules are $\leq \sqcap \mathtt{L}_1$ and $\leq \sqcap \mathtt{L}_2$, both of which get stuck because we lose half the information we require for the rest of the derivation. The situation is perhaps exacerbated by the fact that we *can* prove subtyping in the other direction, so these types are supposed to be equivalent. This means depending on where these types occur, we may fail to prove one side of a symmetric relation!

The fix is not as simple as adding this rule as an extra axiom. For one, it is not trivial to rewrite this rule in order to preserve admissibility of transitivity. More importantly, this is not the only rule we would have to add. Figure 4.1 gives just *some* of the many sound rules that are not admissible.

$$(A_1 \otimes A_2) \sqcap (B_1 \otimes B_2) \leq (A_1 \sqcap B_1) \otimes (A_2 \sqcap B_2)$$

$$\oplus \{lab_k : A_k\}_{k \in I} \sqcap \oplus \{lab_k : B_k\}_{k \in J} \leq \oplus \{lab_k : A_k \sqcap B_k\}_{k \in I \cap J}$$

$$(A \multimap B_1) \sqcap (A \multimap B_2) \leq A \multimap (B_1 \sqcap B_2)$$

$$\& \{lab_k : A_k\}_{k \in I} \sqcap \& \{lab_k : B_k\}_{k \in J} \leq \& \{lab_k : A_k\}_{k \in I} \cup \& \{lab_k : B_k\}_{k \in J} \ (I \cap J = \emptyset)$$

$$(A_1 \sqcup A_2) \otimes B \leq (A_1 \otimes B) \sqcup (A_2 \otimes B)$$

$$\oplus \{lab_k : A_k\}_{k \in I} \cup \oplus \{lab_k : B_k\}_{k \in J} \leq \oplus \{lab_k : A_k\}_{k \in I} \sqcup \oplus \{lab_k : B_k\}_{k \in J} \ (I \cap J = \emptyset)$$

$$(A_1 \multimap B) \sqcap (A_2 \multimap B) \leq (A_1 \sqcup A_2) \multimap B$$

$$\& \{lab_k : A_k \sqcup B_k\}_{k \in I \cap J} \leq \& \{lab_k : A_k\}_{k \in I} \sqcup \& \{lab_k : B_k\}_{k \in J}$$

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

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

FIGURE 4.1: Sound but inadmissible subtyping rules

This list is certainly not complete, and there are almost as many rules in this list as there are in the original system. Clearly, a blind axiomatic approach is not practical and a more general treatment is in order. There has been some work on incorporating intersections and unions in a conventional type system that preserves completeness under certain conditions. The closest and most complete system we found was from Damm [7, 6]. In [7], he encodes types as regular tree expressions, and reformulates subtyping as regular tree grammar containment which was shown to be decidable. This results in a

system that is sound and complete when all types are infinite (but **1** is a finite type). He later extends this work [6] such that the system is sound in the presence of finite types (although not necessarily complete).

Their system is very close to what we would like to accomplish, however, we think it is too complicated for our purposes as it requires familiarity with ideal semantics of types (which in turn is based on domain theory and the theory metric spaces) and regular tree grammars. Even if a similar approach could be made to work, we find such systems to be fragile in the face of future extensions. We would rather work with a simple and robust subtyping relation that does not necessitate rethinking every detail with every extension, so we make a design decision: we give up full completeness and instead design a system that admits the rules we are likely to encounter in practice.

In the next section, we present a system that we think achieves the right tradeoff between simplicity and completeness.

4.3.2 Subtyping as Sequent Calculus with Multiple Conclusions

Since we are interested in intersections and unions, we would like to at least admit distributivity of intersection over union and vice versa. That is, we would like the following equalities to hold:

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

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

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 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). This results in a system reminiscent of classical sequent calculus with multiple conclusions [13, 14]. However, our system is slightly different since we are working with coinductive rules.

[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.

The new system is presented Figure 4.2. 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. This results is a system that does not admit distributivity of intersections and unions over structural types. We conjecture that matching multiple types will give distributivity over structural types, though the intricacies that arise has lead us to leave this to future work.

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

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

$$\frac{}{\alpha, \mathbf{1} \Rightarrow \mathbf{1}, \beta} \Rightarrow \mathbf{1} \qquad \frac{A \Rightarrow A' \quad B \Rightarrow B'}{\alpha, A \otimes B \Rightarrow A' \otimes B', \beta} \Rightarrow \otimes$$

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

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

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

FIGURE 4.2: Subtyping with multiple hypothesis and conclusions

4.3.3 Properties of Subtyping

Remark 4.1. Note that the new system in fact admits the desired distributivity rules. That is, we have $(A_1 \sqcup B) \sqcap (A_2 \sqcup B) \Rightarrow (A_1 \sqcap A_2) \sqcup B$ and $(A_1 \sqcup A_2) \sqcap B \Rightarrow (A_1 \sqcap B) \sqcup (A_2 \sqcap B)$.

Proof. We give the following derivation for the former:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\texttt{id}(A_1)}{A_1 \Rightarrow A_1}}{A_1, A_2 \sqcup B \Rightarrow A_1, B}\,\texttt{weak} \quad \dfrac{\vdots}{B, A_2 \sqcup B \Rightarrow A_1, B}}{A_1 \sqcup B, A_2 \sqcup B \Rightarrow A_1, B} \Rightarrow \sqcup\texttt{L} \quad \dfrac{\vdots}{A_1 \sqcup B, A_2 \sqcup B \Rightarrow A_2, B}}{A_1 \sqcup B, A_2 \sqcup B \Rightarrow A_1 \sqcap A_2, B} \Rightarrow \sqcap\texttt{R}}{(A_1 \sqcup B) \sqcap (A_2 \sqcup B) \Rightarrow A_1 \sqcap A_2, B} \Rightarrow \sqcap\texttt{L}}{(A_1 \sqcup B) \sqcap (A_2 \sqcup B) \Rightarrow (A_1 \sqcap A_2) \sqcup B} \Rightarrow \sqcup\texttt{R}$$

where `weak` and `id` are the weakening and identity lemmas proven later in this section. The proof of the latter relation is similar. □

Intuitively, these proofs go thorough since $\Rightarrow \sqcap\texttt{L}$ and $\Rightarrow \sqcup\texttt{R}$ do not lose information. Formally, this means these rules are invertible:

Lemma 4.2 (Invertibility)**.** *The following are admissible:*

$(\Rightarrow \sqcap\texttt{R})$ $\alpha \Rightarrow A_1 \sqcap A_2, \beta \iff \alpha \Rightarrow A_1, \beta$ *and* $\alpha \Rightarrow A_2, \beta$.

$(\Rightarrow \sqcap\texttt{L})$ $\alpha, A_1 \sqcap A_2 \Rightarrow \beta \iff \alpha, A_1, A_2 \Rightarrow \beta$.

$(\Rightarrow \sqcup\texttt{R})$ $\alpha \Rightarrow A_1 \sqcup A_2, \beta \iff \alpha \Rightarrow A_1, A_2, \beta$.

$(\Rightarrow \sqcup\texttt{L})$ $\alpha, A_1 \sqcup A_2 \Rightarrow \beta \iff \alpha, A_1 \Rightarrow \beta$ *and* $\alpha, A_2 \Rightarrow \beta$.

$(\Rightarrow \mu\texttt{R})$ $\alpha \Rightarrow \mu t.A_t, \beta \iff \alpha \Rightarrow [\mu t.A_t/t]A_t, \beta$.

$(\Rightarrow \mu\texttt{L})$ $\alpha, \mu t.A_t \Rightarrow \beta \iff \alpha, [\mu t.A_t/t]A_t \Rightarrow \beta$.

Proof. Right to left follows by an immediate application of $\Rightarrow \sqcap\texttt{R}$, $\Rightarrow \sqcap\texttt{L}$, $\Rightarrow \sqcup\texttt{R}$, $\Rightarrow \sqcup\texttt{L}$, $\Rightarrow \mu\texttt{R}$, or $\Rightarrow \sqcup\texttt{L}$ respectively.

For the other direction, we proceed with coinduction on the derivation of the subtyping judgment. Call the type we care about A.

- If the proof is by a structural rule ($\Rightarrow \mathbf{1}$, $\Rightarrow \otimes$, $\Rightarrow \oplus$, $\Rightarrow \multimap$, or $\Rightarrow \&$), it must be on types in α and β, so we use the same rule with the same premises to prove the result.
- If the proof is by the relevant rule on A, then the premises are exactly what we need.
- Otherwise, the proof deconstructs a type in α or β using a property rule without changing A. Using the same rule and the coinduction hypotheses gives the result.

□

One might think the cases for $\Rightarrow \sqcap\mathtt{L}$ and $\Rightarrow \sqcup\mathtt{R}$ can be strengthened. After all, if $\alpha \Rightarrow A_1 \sqcup A_2$ then it should be the case that either $\alpha \Rightarrow A_1$ or $\alpha \Rightarrow A_2$. However, this intuition is not true in general. Consider how the proof for $A_1 \sqcup A_2 \Rightarrow A_1 \sqcup A_2$ would proceed: we would apply $\Rightarrow \sqcup\mathtt{L}$ which splits the derivation in half, and then depending on the branch, we would either pick A_1 or A_2 on the right. Luckily, it turns out that we can recover this result if all types on the left are *structural*.

Define A `structural` if A has a structural type construct at the top level (i.e. not μ, $\sqcap$ or $\sqcup$). We say α `structural` or Ψ `structural` if A `structural` for all $A \in \alpha$ or $A \in \Psi$, respectively. Then,

Lemma 4.3 (Extended Invertibility)**.** *The following are admissible:*

- *If* $\alpha \Rightarrow \beta$ *and* α `structural`*, then* $\bigvee_{A \in \beta} \alpha \Rightarrow A$*.*
- *If* $\alpha \Rightarrow \beta$ *and* β `structural`*, then* $\bigvee_{A \in \alpha} A \Rightarrow \beta$*.*

Proof. We will only show the first case; the other is symmetric. We proceed by induction on size β and case analysis on $\mathcal{D} : \alpha \Rightarrow \beta$.

- $\mathcal{D}$ is by $\Rightarrow \sqcap\mathtt{R}$. The induction hypotheses give $\bigvee_{A \in B_1, \beta'} \alpha \Rightarrow A$ and $\bigvee_{A \in B_2, \beta'} \alpha \Rightarrow A$ where $\beta = B_1 \sqcap B_2, \beta'$. If $\alpha \Rightarrow B_1$ and $\alpha \Rightarrow B_2$ then $\alpha \Rightarrow B_1 \sqcap B_2$ by $\Rightarrow \sqcap\mathtt{R}$. Otherwise, $\bigvee_{A \in \beta'} \alpha \Rightarrow A$.
- $\mathcal{D}$ is by $\Rightarrow \sqcup\mathtt{R}$. The induction hypothesis gives $\bigvee_{A \in B_1, B_2, \beta'} \alpha \Rightarrow A$ where $\beta = B_1 \sqcup B_2, \beta'$. If $\alpha \Rightarrow B_1$ or $\alpha \Rightarrow B_2$, then $\alpha \Rightarrow B_1, B_2$ by `weak`, and thus $\alpha \Rightarrow B_1 \sqcup B_2$ by $\Rightarrow B_1 \sqcup B_2$. Otherwise, we have $\bigvee_{A \in \beta'} \alpha \Rightarrow A$.
- $\mathcal{D}$ is by $\Rightarrow \mu\mathtt{R}$. The induction hypothesis gives $\bigvee_{A \in [\mu t.B_t/t]B_t, \beta'} \alpha \Rightarrow A$ where $\beta = \mu t.B_t, \beta'$. If $\alpha \Rightarrow \mu t.B_t t B_t$, then the result follows form $\mu\mathtt{R}$. Otherwise, it is immediate.
- $\mathcal{D}$ is by by a structural rule. The rule picks one type in β and the rest are irrelevant, so the result is immediate.

The other cases are impossible since α `structural`. □

Corollary 4.4. *We have the following:*

- *If* $\alpha \Rightarrow A_1 \sqcup A_2$ *and* α `structural`*, then either* $\alpha \Rightarrow A_1$ *or* $\alpha \Rightarrow A_2$*.*

- *If* $A_1 \sqcup A_2 \Rightarrow \beta$ *and* β `structural`*, then either* $A_1 \Rightarrow \beta$ *or* $A_2 \Rightarrow \beta$*.*

Proof. Follows trivially from inversion and lemma 4.3. □

Next, we verify that usual properties expected of logical systems (such as weakening and cut admissibility) are valid for our system.

Lemma 4.5 (Weakening)**.** *If* $\alpha \Rightarrow \beta$*, then* $\alpha, \alpha' \Rightarrow \beta, \beta'$ *for all* α', β'*.*

Proof. Follows from a trivial coinduction on the derivation of $\alpha \Rightarrow \beta$ since all rules are parametric on the unused types. □

Theorem 4.6 (Admissibility of Identity)**.** $\alpha \Rightarrow \alpha$ *for all non-empty* α*.*

Proof. We will prove a more general statement which says $\alpha', A \Rightarrow A, \beta'$ for any A, α', β'. Since α in question is non-empty, we can instantiate A to be any type in α and pick $\alpha' = \beta'$ to be the rest of the list.

We proceed by coinduction on the structure of A.

- If A has a structural construct at the top level, we apply the corresponding rule for $\Rightarrow$ and satisfy the premises using the coinduction hypotheses.
- If A is a μ, we unfold on both sides and use the coinduction hypothesis.
- If A is a $\sqcap$ or a $\sqcup$, we unfold on both sides. The derivation splits into two cases, which we satisfy using the coinduction hypotheses.

□

Theorem 4.7 (Admissibility of Cut)**.** *If* $\alpha \Rightarrow A, \beta$ *and* $\alpha, A \Rightarrow \beta$*, then* $\alpha \Rightarrow \beta$*.*

Proof. Assume $\mathcal{D} : \alpha \Rightarrow A, \beta$ and $\mathcal{E} : \alpha, A \Rightarrow \beta$. We will use a lexicographic combination of coinduction on $\mathcal{D}$ and $\mathcal{E}$ and induction on size A to construct a proof of $\mathcal{F} : \alpha \Rightarrow \beta$:

- If $\mathcal{D}$ is by a structural rule not using A, then the same rule with the same premises proves $\mathcal{F}$. We make no use of $\mathcal{E}$.
- Similarly, if $\mathcal{E}$ is by a structural rule not using A, then the same rule with the same premises proves $\mathcal{F}$. We do not need $\mathcal{D}$ in this case.

- $\mathcal{D}$ is by a non-structural rule not using A. We will only show the case for $\Rightarrow \sqcap\mathtt{L}$ since other cases are similar. We have:

$$\mathcal{D} = \frac{\mathcal{D}' : \alpha', B_1, B_2 \Rightarrow A, \beta}{\alpha', B_1 \sqcap B_2 \Rightarrow A, \beta} \Rightarrow \sqcap\mathtt{L}$$

  Lemma 4.2 on $\mathcal{E}$ gives $\mathcal{E}' : \alpha', B_1, B_2, A \Rightarrow \beta$. Then, we can construct $\mathcal{F}$ as follows:

$$\frac{\mathtt{cut}(\mathcal{D}', \mathcal{E}', A)}{\alpha', B_1, B_2 \Rightarrow \beta} \Rightarrow \sqcap\mathtt{L}$$

- $\mathcal{E}$ is by a non-structural rule not using A. Similar to the previous case.

- $\mathcal{D}$ and $\mathcal{E}$ are by the same structural rule on A. We will only present the case for $\Rightarrow \multimap$. We have:

$$\mathcal{D} = \frac{\mathcal{D}_1 : A_1 \Rightarrow B_1 \quad \mathcal{D}_2 : B_2 \Rightarrow A_2}{\alpha', B_1 \multimap B_2 \Rightarrow A_1 \multimap A_2, \beta} \Rightarrow \multimap$$

$$\mathcal{E} = \frac{\mathcal{E}_1 : C_1 \Rightarrow A_1 \quad \mathcal{E}_2 : A_2 \Rightarrow C_2}{\alpha, A_1 \multimap A_2 \Rightarrow C_1 \multimap C_2, \beta'} \Rightarrow \multimap$$

  We construct $\mathcal{F}$ as follows:

$$\frac{\begin{array}{cc}\mathtt{trans}(\mathcal{E}_1, \mathcal{D}_1, A_1) & \mathtt{trans}(\mathcal{D}_2, \mathcal{E}_2, A_2) \\ C_1 \Rightarrow B_1 & B_2 \Rightarrow C_2\end{array}}{\alpha', B_1 \multimap B_2 \Rightarrow C_1 \multimap C_2, \beta'} \Rightarrow \multimap$$

  where `trans` is a specific use of `cut` defined in corollary 4.8.

- $\mathcal{D}$ and $\mathcal{E}$ are by symmetric non-structural rules on A. This turns out to be the most tricky case and the only case where we need the induction. Case for $\mu\mathtt{R}/\mu\mathtt{L}$ follows from an immediate application of the coinduction hypothesis (note that size (A) shrinks by one). Cases for $\sqcap\mathtt{R}/\sqcap\mathtt{L}$ and $\sqcup\mathtt{R}/\sqcup\mathtt{L}$ are similar, so we will show the former. We have:

$$\mathcal{D} = \frac{\mathcal{D}_1 : \alpha \Rightarrow A_1, \beta \quad \mathcal{D}_2 : \alpha \Rightarrow A_2, \beta}{\alpha \Rightarrow A_1 \sqcap A_2, \beta} \Rightarrow \sqcap\mathtt{R} \qquad \mathcal{E} = \frac{\mathcal{E}' : \alpha, A_1, A_2 \Rightarrow \beta}{\alpha, A_1 \sqcap A_2 \Rightarrow \beta} \Rightarrow \sqcap\mathtt{L}$$

  We then define F as follows:

$$\dfrac{\mathcal{D}_1}{\alpha \Rightarrow A_1, \beta} \quad \dfrac{\dfrac{\mathcal{D}_2}{\alpha, A_1 \Rightarrow A_2, \beta}\,\mathtt{weak} \quad \dfrac{\mathcal{E}'}{\alpha, A_1, A_2 \Rightarrow \beta}}{\alpha, A_1 \Rightarrow \beta}\,\mathtt{cut}}{\alpha \Rightarrow \beta}\,\mathtt{cut}$$

Note that the size of the type is inductively smaller than size (A) in every (co)inductive application of `cut`.

□

Of course, we do not have multisets of types at the top level. The final subtyping relation is thus defined by passing singleton sets to the relation: "A is a subtype of B" $\triangleq A \Rightarrow B$. It makes sense to ask whether the relation on singleton sets is a preorder, which easily follows from what we have proved:

Corollary 4.8. $\Rightarrow$ *is a preorder:*

- $A \Rightarrow A$ *for all types* A.
- $A \Rightarrow B$ *and* $B \Rightarrow C$ *implies* $A \Rightarrow C$ *for all types* A, B, C.

Proof. Reflexivity is a special case of identity (theorem 4.6). Transitivity follows from cut admissibility using weakening: assume $A \Rightarrow B$ and $B \Rightarrow C$. By weakening, $A \Rightarrow B, C$ and $A, B \Rightarrow C$. This now matches the form required by theorem 4.7. □

4.3.4 Completeness of $\Rightarrow$ with Respect to $\leq$

Whenever one changes the structure of a judgment, it is a good idea to establish a relation between the new and the old judgments. Usually, the relation established is equivalence, where the new system is shown to be sound and complete with respect to the old. Completeness shows that the new system is not any weaker: everything derivable in the old system is also derivable in the new. Soundness is the converse: everything derivable in the new system was already derivable in the old. Soundness clearly fails here since $\Rightarrow$ admits distributivity among property types whereas $\leq$ does not. In fact, this was precisely the reason for switching over! We still would like to make sure that we are not *losing* and relations, which is witnessed by the following theorem:

Theorem 4.9 (Soundness of $\Rightarrow$)**.** *If* $A \leq B$ *then* $A \Rightarrow B$.

Proof. By a straightforward coinduction on the derivation of $A \leq B$. In each case, we apply the corresponding rule for $\Rightarrow$, using lemma 4.5 to weaken the premises of $\Rightarrow \sqcap\mathtt{L}$ and $\Rightarrow \sqcup\mathtt{R}$ as necessary. □

4.4 Encoding n-ary Choice Using Intersections and Unions

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. We do this by reinterpreting n-ary choices using intersections and unions, in a similar manner to Reynolds' reinterpretation of records using intersections [19]. 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: $\&\{\texttt{inl} : A, \texttt{inr} : B\}$. 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 external 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 verified that these definitions satisfy the typing and subtyping rules for external and internal choices, so they are faithful to the meaning of original constructs. At this point, we could remove n-ary external and internal choices from the system in favor of singleton choices. These definitions do not indicate what happens when I is empty, however. We could either insist this be not the case (since there are not many useful programs that make use of empty choices), or we could add empty choices along with singleton choices.

An advantage of singleton choices is that they simplify the typing rules:

$$\frac{\Psi \vdash_\eta P :: (c : A)}{\Psi \vdash_\eta c.lab\,;\, P :: (c : \oplus\{lab : A\})}\ \oplus\mathtt{R}$$

$$\frac{i \in I \quad \Psi, c : A \vdash_\eta P_i :: (d : D)}{\Psi, c : \oplus\{lab_i : A\} \vdash_\eta \text{case } c \text{ of } \{lab_k \to P_k\}_{k \in I} :: (d : D)}\ \oplus\mathtt{L}$$

$$\frac{i \in I \quad \Psi \vdash_\eta P_i :: (c : A)}{\Psi \vdash_\eta \text{case } c \text{ of } \{lab_k \to P_k\}_{k \in I} :: (c : \&\{lab_i : A\})}\ \&\mathtt{R}$$

$$\frac{\Psi, c : A \vdash_\eta P :: (d : D)}{\Psi, c : \&\{lab : A\} \vdash_\eta c.lab\,;\, P :: (d : D)}\ \&\mathtt{L}$$

and the subtyping rules:

$$\frac{A \leq A'}{\oplus\{lab : A\} \leq \oplus\{lab : A'\}} \leq \oplus \qquad \frac{A \leq A'}{\&\{lab : A\} \leq \&\{lab : A'\}} \leq \&$$

Another advantage is we recover some form of distributivity of intersections and unions over choices. In particular, the following relations hold simply by definition (modulo the commutativity and associativity of intersection and union):

$$\&\{lab_k : A_k\}_{k\in I} \sqcap \&\{lab_k : B_k\}_{k\in J} \leq \&\{lab_k : A_k\}_{k\in I} \cup \&\{lab_k : B_k\}_{k\in J} \ (I \cap J = \emptyset)$$

$$\oplus \{lab_k : A_k\}_{k\in I} \cup \oplus\{lab_k : B_k\}_{k\in J} \leq \oplus\{lab_k : A_k\}_{k\in I} \sqcup \oplus\{lab_k : B_k\}_{k\in J} \ (I \cap J = \emptyset)$$

Unfortunately, some properties still do not hold since we do not have distributivity over structural types. For example, we cannot derive the following in general:

$$\oplus\{lab_k : A_k\}_{k\in I} \sqcap \oplus\{lab_k : B_k\}_{k\in J} \leq \oplus\{lab_k : A_k \sqcap B_k\}_{k\in I\cap J}$$

$$\&\{lab_k : A_k \sqcup B_k\}_{k\in I\cap J} \leq \&\{lab_k : A_k\}_{k\in I} \sqcup \&\{lab_k : B_k\}_{k\in J}$$

Still, this encoding simplifies the type system and establishes the connection between intersections and external choice, and unions and internal choice.

4.5 Type Safety

It turns out type preservation fails with the current formulation of the system. The problem stems from the fact that `cut`, $\otimes$R, and $\multimap$ L split the context into two, however, these splits do not have to be unique. In the base system, the split is only done once so type safety holds, but $\sqcap$R and $\sqcup$L rules branch into two derivations, each of which could split the context differently. This sort of dependence does not work for configuration typing which must have a tree structure (where the children of a process correspond to all the channels used by that process) *irrespective* of the types of the processes. That is, we need to know exactly which channels go into typing the newly spawned process when we step a cutting or a sending process, and this choice cannot depend on the type.

We will suggest a fix to this problem, but first, let us look at a concrete example which shows preservation fails. The outline of the disproof is as follows. First, we give a process which offers an intersection and a typing derivation where the context is split in two different ways for the two branches of the intersection. Next, we put this process in a configuration and show that it is well typed. Finally, we take a valid step, and argue why the new configuration cannot be well-typed.

Example 4.1. *We will show* $d_1 : \mathbf{1}, d_2 : \mathbf{1} \vdash_\emptyset P :: (c : A)$ *where*

$$A = (A_l \otimes A_r) \sqcap (A_r \otimes A_l) \qquad A_l = \&\{\texttt{inl} : \mathbf{1}\} \qquad A_r = \&\{\texttt{inr} : \mathbf{1}\}$$

$$P = \text{send } c\ (x \leftarrow P'_x)\,;\ P'_c$$

$$P'_x = \text{case } x \text{ of } \{\texttt{inl} \to \text{wait } d_1\,;\ \text{close } x,\ \ \texttt{inr} \to \text{wait } d_2\,;\ \text{close } x\}$$

The typing derivation is given below:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset \vdash_\emptyset \text{close } x :: (x : \mathbf{1})}\,\mathbf{1}R}{d_1 : \mathbf{1} \vdash_\emptyset \text{wait } d_1\,;\ \text{close } x :: (x : \mathbf{1})}\,\mathbf{1}L}{d_1 : \mathbf{1} \vdash_\emptyset P'_x :: (x : A_l)}\,\&R \qquad \begin{array}{c}\vdots\\ d_2 : \mathbf{1} \vdash_\emptyset P'_c :: (c : A_r)\end{array}}{d_1 : \mathbf{1}, d_2 : \mathbf{1} \vdash_\emptyset P :: (c : A_l \otimes A_r)}\,\otimes R \qquad \cdots}{d_1 : \mathbf{1}, d_2 : \mathbf{1} \vdash_\emptyset P :: (c : A)}\,\sqcap R$$

(cases left out are similar). It is important to note that for the left branch, d_1 *is used to type* P'_x *and* d_2 *is used to type* P'_c*, whereas for the right branch, the opposite is true.*

Next, it is easy to check $c : A \vdash_\emptyset Q :: (e : (A_l \otimes A_r \otimes \mathbf{1}) \sqcap (A_r \otimes A_l \otimes \mathbf{1}))$ *where*

$$Q = y \leftarrow \text{recv } c\,;\ \text{send } e\ y\,;\ \text{send } e\ c\,;\ \text{close } e.$$

Now, consider $\Omega = \texttt{proc}_{d_1}(\text{close } d_1), \texttt{proc}_{d_2}(\text{close } d_2), \texttt{proc}_c(P), \texttt{proc}_e(Q)$*, which is easily typed given the derivations above.* $\texttt{proc}_c(P)$ *and* $\texttt{proc}_e(Q)$ *can take a step together using* ***tensor****, but the new configuration is untypeable. The process spawned off by* $\texttt{proc}_c(P)$ *needs both* d_1 *and* d_2*, so does the process* $\texttt{proc}_c(P)$ *steps into. There is no way to split the context properly at the configuration level, thus preservation fails.*[2]

One way to restore type safety is to ensure there is always a unique split. One might be mislead to think this should already be the case: whenever we have $\Psi \vdash_\eta P :: (c : A)$, it must be the case that $\text{free}\, P \setminus \{c\} \subseteq \text{dom}\, \Psi$ since the context must provide all unbound variables, and $\text{dom}\, \Psi \subseteq \text{free}\, P \setminus \{c\}$ since all channels must be used in a linear context. That is, we must have $\text{dom}\, \Psi = \text{free}\, P \setminus \{c\}$ which would uniquely determine the context based on the free channels of a process. However, this fails for three reasons:

1. Empty choices $\&\{\}$ and $\oplus\{\}$ might be well-typed under any context since $\&\mathtt{R}$ and $\oplus\mathtt{L}$ have no premises. For example, we have $\Psi, d : \oplus\{\} \vdash_\emptyset \text{case } d \text{ of } \{\} :: (c : A)$ for any Ψ and A by an application of $\oplus\mathtt{L}$. In this case, we might have $\text{free}\, P \setminus \{c\} \subsetneq \text{dom}\, \Psi$.

[2] Rigorously proving that the new configuration is untypeable is not easy in the declarative system, so we only present an intuitive argument. The proof could be formalized using the system given in section 5.2 if necessary.

2. We allow unused branches in case expressions, which might contain any channel name, so it may not be the case that $\operatorname{dom}\Psi \subsetneq \operatorname{free} P \setminus \{c\}$. This is what we exploited in the counterexample above.

3. Recursive processes may be well-typed in any context. For example:

$$\dfrac{\dfrac{\Psi \vdash p(\emptyset) :: (c : A) \in \{\Psi \vdash p(\emptyset) :: (c : A)\}}{\Psi \vdash_\eta p\ \emptyset :: (c : A)}\ \texttt{var} \qquad \eta = \Psi \vdash p(\emptyset) :: (c : A)}{\Psi \vdash_\emptyset (\operatorname{rec} p(\emptyset).(p\ \emptyset))\ \emptyset :: (c : A)}\ \mu$$

   holds for any Ψ and A.

We introduce the following (reasonable) restrictions to restore this property:

1. Empty choices are disallowed. Whenever we see $\&\{lab_k : A_k\}_{k\in I}$ or $\oplus\{lab_k : A_k\}_{k\in I}$, we check that $I \neq \emptyset$.

2. All branches of a case expressions must have the same set of free channels. That is, if we have case c of $\{lab_k \to P_k\}_{k\in I}$, then we must have $\operatorname{free} P_i = \operatorname{free} P_j$ for $i, j \in I$.

3. In the rule μ, which types $\Psi \vdash_\eta (\operatorname{rec} p(\bar{y}).P)\ \bar{z} :: (c : A)$, we add the premise $\bar{z} = \operatorname{dom}\Psi \cup \{c\}$ so that all relevant channels are explicitly mentioned in the term.

With these restrictions, we get the following regularity theorem:

Theorem 4.10 (Regularity of Contexts)**.** *If* $\Psi \vdash_\eta P :: (c : A)$ *and for all* $\Psi' \vdash p(\bar{y}) :: (d : D) \in \eta$, $\bar{y} = \operatorname{dom}\Psi' \cup \{d\}$, *then* $\operatorname{free} P = \operatorname{dom}\Psi \cup \{c\}$.

Proof. By induction on the typing derivation. The only nontrivial cases are $\&\mathtt{R}$, $\oplus\mathtt{L}$, μ, and $\mathtt{var}$.

Case $\&\mathtt{R}, \oplus\mathtt{L}$: Restriction (1) ensures there is at least one premise on which we can apply the induction hypothesis. Due to restriction (2), it does not matter which branch is typed. The result then follows easily.

Case μ : We know $P = (\operatorname{rec} p(\bar{y}).P')\ \bar{z}$ and $\Psi \vdash_{\eta'} [\bar{z}/\bar{y}]P :: (c : A)$, where $\eta' = \eta, [\bar{y}/\bar{z}]\Psi \vdash p(\bar{y}) :: ([\bar{y}/\bar{z}]c : A)$. In addition, restriction (3) gives $\bar{z} = \operatorname{dom}\Psi \cup \{c\}$. Since $\operatorname{free} P = (\operatorname{free} P' \setminus \bar{y}) \cup \bar{z}$, it suffices to show $\operatorname{free} P' = \bar{y}$.

Before we can apply the induction hypothesis, we need to show $\bar{y} = \operatorname{dom}([\bar{y}/\bar{z}]\Psi) \cup \{[\bar{y}/\bar{z}]c\}$, which follows easily from the fact that $\bar{z} = \operatorname{dom}\Psi \cup \{c\}$. The induction hypothesis then gives $\operatorname{free}([\bar{z}/\bar{y}]P') = \operatorname{dom}\Psi \cup \{c\}$. Finally, we have:

$$\operatorname{free} P' = \operatorname{free}([\bar{y}/\bar{z}][\bar{z}/\bar{y}]P') = \operatorname{dom}([\bar{y}/\bar{z}]\Psi) \cup \{[\bar{y}/\bar{z}]c\} = \bar{y}$$

from before, as required.

Case `var` : We have $P = p\ \bar{z}$, $\Psi = [\bar{z}/\bar{y}]\Psi'$, and $c = [\bar{z}/\bar{y}]d$ where $\Psi' \vdash p(\bar{y}) :: (d : A) \in \eta$. By assumption, $\bar{y} = \operatorname{dom}\Psi' \cup \{d\}$, so

$$\operatorname{free} P = \bar{z} = [\bar{z}/\bar{y}]\bar{y} = \operatorname{dom}([\bar{z}/\bar{y}]\Psi') \cup \{[\bar{z}/\bar{y}]d\} = \operatorname{free}\Psi \cup \{c\}$$

as desired.

□

Corollary 4.11 (Regularity of Contexts)**.** *If* $\Psi \vdash_\emptyset P :: (c : A)$ *then* $\operatorname{dom}\Psi = \operatorname{free} P \setminus \{c\}$.

Proof. Follows trivially from theorem 4.10. The premises are immediately satisfied since the context for process variables is empty, and we can subtract $\{c\}$ from both sides to get the final result. □

It should be worthwhile to talk about why the restrictions above are reasonable. Example 4.1 shows that restriction (2) or a similar restriction on case expression is necessary for type preservation to hold. In addition, it is reasonable to expect all branches of a case expression to be related in a certain way. We could force the programmer to provide a type for all branches even when they are irrelevant, but this would make programming cumbersome and disallow the encoding in section 4.4. Restriction (2) is much easier to satisfy, and it makes sense in a linear setting.

Restrictions (1) and (3) are necessary for theorem 4.10, but we conjecture they are not needed for type preservation. Without (1) and (3), context splits become nondeterministic, so different branches of a derivation could split the context in different ways but not in an essential way.[3] However, the analysis becomes unnecessarily complicated when we remove these restrictions. $\&\{\}$ has no right rule and $\oplus\{\}$ has no left rule, so these types will not come up in real programs. Restriction (3) amounts to labeling each recursive process with the channels in the context, which should always be known. The channels that the programmer chose not to rename can always be filled in by a compiler, so (3) poses no practical concerns.

Corollary 4.11 is sufficient to prove type safety. However, applicability of subsumption at any point in the presence of intersections and unions complicates the proof too much. Instead, we will first introduce the better behaved algorithmic system and establish safety for that system. The equivalence of the algorithmic system to the declarative will then imply safety for the declarative system.

[3] For example, empty choices might be well-typed under any context. Thus, if they are typed under two different contexts Ψ_1 and Ψ_2, they can also be typed under $\Psi_1 \cap \Psi_2$ or $\Psi_1 \cup \Psi_2$.

Chapter 5

An Algorithmic System

In this chapter, 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. It turns out to be easy to convert the subtyping relation we presented in Figure 4.2 into an algorithm. For type-checking, we have to design an *algorithmic system* and prove it equivalent to the original. Then, the decidability of the new system will imply decidability of the old.

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 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 (in any order) since these are invertible (lemma 4.2). When we hit all structural types (which will always be the case due to our contractiveness restriction, though the proof of termination does not depend on this fact) 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.[1] For example, to show

[1] Assuming the conclusion and showing productivity corresponds to how coinductive proofs usually proceed.

$\mu t.1 \multimap t \Rightarrow \mu s.1 \multimap 1 \multimap s$, we could have the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\ldots \vdash 1 \Rightarrow 1}\Rightarrow \mathbf{1} \quad \dfrac{}{\multimap \vdash \mu t.1 \multimap t \Rightarrow \mu s.1 \multimap 1 \multimap s}\,\texttt{hyp}}{\ldots \vdash 1 \multimap \mu t.1 \multimap t \Rightarrow 1 \multimap \mu s.1 \multimap 1 \multimap s}\Rightarrow \multimap \quad \dfrac{}{\ldots \vdash 1 \Rightarrow 1}\Rightarrow \mathbf{1}}{\ldots \vdash \mu t.1 \multimap t \Rightarrow 1 \multimap \mu s.1 \multimap 1 \multimap s}\Rightarrow \mu\mathtt{L}}{\ldots \vdash 1 \multimap \mu t.1 \multimap t \Rightarrow 1 \multimap 1 \multimap \mu s.1 \multimap 1 \multimap s}\Rightarrow \multimap}{(\mu t.1 \multimap t \Rightarrow \mu s.1 \multimap 1 \multimap s) \vdash 1 \multimap \mu t.1 \multimap t \Rightarrow \mu s.1 \multimap 1 \multimap s}\Rightarrow \mu\mathtt{R}}{\emptyset \vdash \mu t.1 \multimap t \Rightarrow \mu s.1 \multimap 1 \multimap s}\Rightarrow \mu\mathtt{L}$$

The contexts are elided (written $\ldots$) to save space, but they are always supersets of $\mu t.1 \multimap t \Rightarrow \mu s.1 \multimap 1 \multimap 1$. `hyp` simply matches the goal with an assumption in the context.

It is not hard to see that the algorithm is a correct decision procedure for the subtyping system. If a subtyping judgment is derivable in the original system, it certainly is derivable with the additional rule since all `hyp` can do is to terminate the derivation at a finite depth. Conversely, we can always convert a proof that uses `hyp` into a coinductive (and infinite) proof since we must have made progress between the introduction of a hypothesis and its usage. We simply duplicate this partial derivation indefinitely.

Finally, we need to show that the algorithm always terminates. At every step, the context grows by one element.[2] Thus, termination comes down to finding a finite upper bound on the size of the context. Assume we are trying to establish $A \Rightarrow B$. The only types that can appear in the context are substructures of A and B (modulo unfolding) of which there are finitely many. Since subtyping is defined on pairs, and each side of $\Rightarrow$ is a multiset rather than a single type, the maximum size increases exponentially, but still stays finite. A more formal treatment can be found in [21].

5.2 Bidirectional Type-checking

Designing a syntax directed 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 $c : A \leftarrow P_c\,;\, Q_c$. We define $[\![P]\!]$ to be the function which erases these annotations.

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

[2] If it does not, we must be adding an element that already exists, but then we should have applied `hyp`.

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 *bidirectional type-checking* where we only check subtyping at the identity rule and at recursive process variables (delegation). This relies on the subformula property for the sequent calculus, excepting only the cut rule which is annotated. The fact that we can delay subtyping in this way is formalized later in lemma 5.10, when we prove the equivalence of the algorithmic system to the declarative one.

Algorithmic typing rules for processes are given in Figure 5.1. The rules make use of the following definitions:

Definition 5.1 (Cumulative Intersection and Union)**.** For all non-empty $\alpha = A_1, \ldots, A_n$, define $\bigsqcap \alpha = A_1 \sqcap \ldots \sqcap A_n$ and $\bigsqcup \alpha = A_1 \sqcup \ldots \sqcup A_n$. Even though we consider α to be unordered, these operations are well-defined since $\sqcap$ and $\sqcup$ are associative and commutative (with respect to the subtyping relation).

Similarly, for a context $\Psi = c_1 : \alpha_1, \ldots, c_n : \alpha_n$, define $\bigsqcap \Psi = c_1 : \bigsqcap \alpha_1, \ldots, c_n : \bigsqcap \alpha_n$ and $\bigsqcup \Psi = c_1 : \bigsqcup \alpha_1, \ldots, c_n : \bigsqcup \alpha_n$.

Definition 5.2 (Subtyping of Contexts)**.** For $\Psi = c_1 : \alpha_1, \ldots, c_n : \alpha_n$ and $\Psi' = c_1 : \beta_1, \ldots, c_n : \beta_n$, we say $\Psi \Rightarrow \Psi'$ whenever $\alpha_i \Rightarrow \bigsqcap \beta_i$ for all $1 \leq i \leq n$.

The rules $\sqcap\mathtt{R}$, $\sqcap\mathtt{L}$, $\sqcup\mathtt{R}$, $\sqcup\mathtt{L}$, $\mu\mathtt{R}$ and $\mu\mathtt{L}$ are applied eagerly as mentioned before. If and when these rules are saturated[3], we examine the structure of the process and non-deterministically commit to a type. Whenever we need to split the context, we again try all possible splits.[4] At every step, either the process gets structurally smaller, or it stays the same and the size of a type in the context or the provided type shrinks by 1. Thus, the algorithm always terminates.

[3] size of the context or the provided type decreases by one after each application of these rules. Since we only consider contractive types, size is finite and we must reach all structural types at some point.

[4] One might be tempted to think the correct split can be computed based on the free channels in the components of P. Unfortunately, that approach does not always work since case statements might have unused branches. These extra branches are never type-checked, so they could contain free occurrences of any channel name.

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

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

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

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

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

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

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

$$\frac{\Psi, d : A, c : B \Vdash_\eta P_d :: (e : \beta)}{\Psi, c : (\alpha, A \otimes B) \Vdash_\eta d \leftarrow \text{recv } c\,;\, P_d :: (e : \beta)}\ \otimes\mathtt{L}$$

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

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

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

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

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

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

$$\frac{\Psi \vdash_{\eta'} [\bar{z}/\bar{y}]P :: (c : \alpha) \quad \eta' = \eta, [\bar{y}/\bar{z}]\Psi \vdash p(\bar{y}) :: ([\bar{y}/\bar{z}]c : \alpha)}{\Psi \Vdash_\eta (\text{rec } p(\bar{y}).P)\ \bar{z} :: (c : \alpha)}\ \mu$$

$$\frac{(\Psi \vdash p(\bar{y}) :: (c : \alpha)) \in \eta \quad \Psi' \Rightarrow [\bar{z}/\bar{y}]\Psi \quad \bigsqcup \alpha \Rightarrow \beta}{\Psi' \Vdash_\eta p\ \bar{z} :: ([\bar{z}/\bar{y}]c : \beta)}\ \mathtt{var}$$

FIGURE 5.1: Algorithmic process typing

5.3 Properties of Algorithmic Type-checking

We now explore the properties of the bidirectional system analogous to the ones we proved for the multiset subtyping relation in section 4.3.3. First, we have the following definitions:

Definition 5.3 (Operations on Comtexts)**.** For all contexts Ψ and Ψ', we have:

$$\Psi \cap \Psi' = \{c : \Psi(c) \cap \Psi'(c) \mid c \in \operatorname{dom}\Psi \cap \operatorname{dom}\Psi'\}$$

$$\Psi \cup \Psi' = \{c : \Psi(c), \Psi'(c) \mid c \in \operatorname{dom}\Psi \cap \operatorname{dom}\Psi'\}, (\Psi \setminus \operatorname{dom}\Psi'), (\Psi' \setminus \operatorname{dom}\Psi)$$

$$\Psi \uparrow \bar{c} = \{c : \Psi(c) \mid c \in \operatorname{dom}\Psi \cap \bar{c}\}$$

where intersection and union of multisets is defined as usual.

Lemma 5.4 (Weakening)**.** *If* $\Psi \Vdash_\eta P :: (c : \alpha)$*, then* $\Psi \cup \Psi' \Vdash_\eta P :: (C : \alpha, \alpha')$ *for all* α' *and* Ψ' *such that* $\operatorname{dom}\Psi' \subseteq \operatorname{dom}\Psi$*.*

Proof. By a simple induction on the typing derivation since all rules are parametric in the unused types. □

Lemma 5.5 (Invertibility)**.** *The following are admissible:*

($\sqcap\mathtt{R}$) $\Psi \Vdash_\eta P :: (c : A_1 \sqcap A_2, \alpha) \iff \Psi \Vdash_\eta P :: (c : A_1, \alpha)$ *and* $\Psi \Vdash_\eta P :: (c : A_2, \alpha)$.

($\sqcap\mathtt{L}$) $\Psi, d : \alpha, A_1 \sqcap A_2 \Vdash_\eta P :: (c : \beta) \iff \Psi, d : \alpha, A_1, A_2 \Vdash_\eta P :: (c : \beta)$.

($\sqcup\mathtt{R}$) $\Psi \Vdash_\eta P :: (c : A_1 \sqcup A_2, \alpha) \iff \Psi \Vdash_\eta P :: (c : A_1, A_2, \alpha)$.

($\sqcup\mathtt{L}$) $\Psi, d : \alpha, A_1 \sqcup A_2 \Vdash_\eta P :: (c : \beta) \iff \Psi, d : \alpha, A_1 \Vdash_\eta P :: (c : \beta)$ *and* $\Psi, d : \alpha, A_2 \Vdash_\eta P :: (c : \beta)$.

($\mu\mathtt{R}$) $\Psi \Vdash_\eta P :: (c : \mu t.A, \alpha) \iff \Psi \Vdash_\eta P :: (c : [\mu t.A/t]A, \alpha)$.

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

Proof. Right to left is derivable by an application of $\sqcap\mathtt{R}$, $\sqcap\mathtt{L}$, $\sqcup\mathtt{R}$, $\sqcup\mathtt{L}$, $\mu\mathtt{R}$, or $\mu\mathtt{L}$ respectively. Forward direction is by induction on the typing derivation. □

Just as it was the case for subtyping in lemma 4.3, we get a stronger inversion property for $\sqcap\mathtt{L}$ and $\sqcup\mathtt{R}$ when all other types are structural. However, in addition to this restriction, we need the process to be about to communicate along the channel whose type we are inverting. The theorem is not true if the typing derivation can apply a structural rule on a different channel.

To express this more formally, we introduce a new judgement $\texttt{proc}_c(P)\ \texttt{blocked on}\ d$, which intuitively states P's next action will be along d. The formal definition is given in Figure 5.2. Note that we can then say $\texttt{proc}_c(P)\ \texttt{poised} \iff \texttt{proc}_c(P)\ \texttt{blocked on}\ c$.

$$\frac{}{\texttt{proc}_c(\text{close } c)\ \texttt{blocked on}\ c} \qquad \frac{c \neq d}{\texttt{proc}_c(\text{wait } d\,;\, Q)\ \texttt{blocked on}\ d}$$

$$\frac{}{\texttt{proc}_c(\text{send } d\ (e \leftarrow P_e)\,;\, Q)\ \texttt{blocked on}\ d} \qquad \frac{}{\texttt{proc}_c(x \leftarrow \text{recv } d\,;\, Q_x)\ \texttt{blocked on}\ d}$$

$$\frac{}{\texttt{proc}_c(d.lab_i\,;\, P)\ \texttt{blocked on}\ d} \qquad \frac{}{\texttt{proc}_c(\text{case } d \text{ of } \{lab_k \to Q_k\}_{k \in I})\ \texttt{blocked on}\ d}$$

FIGURE 5.2: Blocking channel

We now have the following lemma.

Lemma 5.6 (Extended Invertibility)**.** *The following are admissible:*

- *If* $\Psi \Vdash_\eta P :: (c : \alpha)$, Ψ `structural`, *and* $\texttt{proc}_c(P)$ `poised`, *then* $\bigvee_{A \in \alpha} \Psi \Vdash_\eta P :: (c : A)$.
- *If* $\Psi, d : \alpha \Vdash_\eta P :: (c : \beta)$, Ψ `structural`, β `structural`, *and* $\texttt{proc}_c(P)\ \texttt{blocked on}\ d$, *then* $\bigvee_{A \in \alpha} \Psi, d : A \Vdash_\eta P :: (c : \beta)$.

Proof. The proof is very similar to the proof of lemma 4.3 so we will not repeat it. □

A special case of the above lemma gives the following:

Corollary 5.7. *The following are admissible:*

- *If* $\Psi \Vdash_\eta P :: (c : A_1 \sqcup A_2)$, Ψ `structural`, *and* $\texttt{proc}_c(P)$ `poised`, *then either* $\Psi \Vdash_\eta P :: (c : A_1)$ *or* $\Psi \Vdash_\eta P :: (c : A_2)$.
- *If* $\Psi, d : A_1 \sqcap A_2 \Vdash_\eta P :: (c : \beta)$, Ψ `structural`, β `structural`, *and* $\texttt{proc}_c(P)\ \texttt{blocked on}\ d$, *then either* $\Psi, d : A_1 \Vdash_\eta P :: (c : \beta)$ *or* $\Psi, d : A_2 \Vdash_\eta P :: (c : \beta)$.

Proof. By inversion on the typing judgement and lemma 5.6. □

Finally, we will need a way to relate the channels in a typing context to the process being typed.

Lemma 5.8. *If* $\Psi \vdash_\eta P :: (c : \alpha)$ *and* $\texttt{proc}_c(P)\ \texttt{blocked on}\ d$ *where* $c \neq d$, *then* $d \in \operatorname{dom} \Psi$.

Proof. Follows from a simple induction on the typing derivation. □

5.4 Equivalence to the Declarative System

We are now ready to prove that the algorithmic system is equivalent (modulo type annotations) to the declarative system. We do this by showing the soundness and completeness of the algorithmic system with respect to the declarative system.

5.4.1 Soundness

Theorem 5.9 (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* $\bigsqcup$*.*

Proof. We proceed by induction on the typing derivation.

Case $\sqcap\mathtt{R}$: This is one of the two tricky cases (the other is the dual $\sqcup\mathtt{L}$). In fact, this case is the reason we needed distributivity of intersection over union. Assume $\mathcal{D} : \Psi \Vdash_\eta P :: (c : A, \alpha)$ and $\mathcal{E} : \Psi \Vdash_\eta P :: (c : B, \alpha)$. We have the following derivation:

$$\dfrac{\dfrac{\begin{array}{cc}\text{inductioHyp}\,(\mathcal{D}) & \text{inductioHyp}\,(\mathcal{E}) \\ \bigsqcap \Psi \Vdash_\eta P :: (c : A \sqcup \bigsqcup \alpha) & \bigsqcap \Psi \Vdash_\eta P :: (c : B \sqcup \bigsqcup \alpha)\end{array}}{\bigsqcap \Psi \Vdash_\eta P :: (c : (A \sqcup \bigsqcup \alpha) \sqcap (B \sqcup \bigsqcup \alpha))}\,\sqcap\mathtt{R} \quad \overset{\text{Remark 4.1}}{\cdots \Rightarrow \cdots}}{\bigsqcap \Psi \Vdash_\eta P :: (c : \bigsqcup (A \sqcap B, \alpha))}\,\mathtt{SubR}$$

where remark 4.1 is used to prove $(A \sqcup \bigsqcup \alpha) \sqcap (B \sqcup \bigsqcup \alpha) \Rightarrow (A \sqcap B) \sqcup \bigsqcup \alpha$.

Case $\sqcap\mathtt{L}$: Follows from `SubL` (we need subtyping to reorder the large intersection) and the induction hypothesis.

Case $\sqcup\mathtt{R}$: Follows from `SubR` (to reorder the large union) and the induction hypothesis.

Case $\sqcup\mathtt{L}$: Similar to $\sqcap\mathtt{R}$.

Case $\mu\mathtt{R}, \mu\mathtt{L}$: Follows immediately from `SubR` and `SubL`, respectively.

Case `id` : Follows by an application of `SubR` and `id`. We note that if $\alpha \Rightarrow \beta$, then $\bigsqcap \alpha \Rightarrow \bigsqcup \beta$ by a trivial induction on the sum of the number of types in α and β.

Case `cut` : Follows immediately from `cut` and the two induction hypotheses.

Case $\mathbf{1}\mathtt{R}, \otimes\mathtt{R}, \oplus\mathtt{R}, \multimap \mathtt{R}, \&\mathtt{R}$: Follows from `SubR` to "pick out the right type" followed by the corresponding rule in the declarative system.

Case $\mathbf{1}\mathtt{L}, \otimes\mathtt{L}, \oplus\mathtt{L}, \multimap \mathtt{L}, \&\mathtt{L}$: Follows from `SubL` to "pick out the right type" followed by the corresponding rule in the declarative system.

Case μ : Follows immediately from the induction hypothesis.

Case `var` : Follows from `SubR`, `SubL`, and the induction hypothesis.

□

5.4.2 Completeness

Lemma 5.10 (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 P stays the same, which means type annotations do not need to change.

Proof. We only show the first part since the two parts are similar. We will use lexicographic induction, first on the structure of P, then on size Ψ + size α + size β. Assume $\mathcal{D} : \Psi \Vdash_\eta P :: (c : \alpha)$ and $\mathcal{E} : \bigsqcup \alpha \Rightarrow \beta$.

- If β contains a non-structural type, we use lemma 4.2 to break it down and apply the induction hypotheses. The result follows from $\sqcap$`R`, $\sqcup$`R`, or μ`R`.
- Otherwise, we know β `structural`, and case on the last rule in $\mathcal{D}$.
  - $\mathcal{D}$ is by $\sqcap$`L`, $\sqcup$`L`, or μ`L`. Follows from the induction hypotheses and the same rule.
  - $\mathcal{D}$ is by $\sqcap$`R`: This is the most interesting case of the proof. We have:

    $$\mathcal{D} = \frac{\mathcal{D}_1 : \Psi \Vdash_\eta P :: (c : A_1, \alpha') \quad \mathcal{D}_2 : \Psi \Vdash_\eta P :: (c : A_2, \alpha')}{\Psi \Vdash_\eta P :: (c : A_1 \sqcap A_2, \alpha')}\ \sqcap\mathtt{R}$$

    By inversion on $\mathcal{E}$, we have $A_1 \sqcap A_2 \Rightarrow \beta$ and $\alpha' \Rightarrow \beta$. Since β `structural`, either $A_1 \Rightarrow \beta$ or $A_2 \Rightarrow \beta$. In the former case, we have $A_1, \alpha' \Rightarrow \beta$, so we can apply the induction hypothesis on $\mathcal{D}_1$ and get the desired result. In the latter, we apply it on $\mathcal{D}_2$.
  - $\mathcal{D}$ is by $\sqcup$`R`: We have $\alpha = A_1 \sqcup A_2, \alpha'$ where $\sqcup$`R` is applied on $A_1 \sqcup A_2$. Since $\bigsqcup(A_1, A_2, \alpha') \Rightarrow \bigsqcup(A_1 \sqcup A_2, \alpha')$, we know $\bigsqcup(A_1, A_2, \alpha') \Rightarrow \beta$. The result follows immediately by the induction hypothesis.
  - $\mathcal{D}$ is by μ`R`: Similar to above since $[\mu t.A_t/t]A_t \Rightarrow \mu t.A_t$.
  - $\mathcal{D}$ is by `id`. Follows from the transitivity of $\Rightarrow$.

– $\mathcal{D}$ is by `cut`. We have

$$\mathcal{D} = \frac{\mathcal{D}_1 : \Psi_1 \Vdash_\eta Q :: (d : \alpha) \quad \mathcal{D}_2 : \Psi_2, d : B \Vdash_\eta Q :: (d : \alpha)}{\Psi_1, \Psi_2 \Vdash_\eta d : B \leftarrow Q \,;\, P' :: (c : \alpha)}\ \texttt{cut}$$

The induction hypothesis on $\mathcal{D}_2$ and $\mathcal{E}$ gives $\Psi_2, d : A \Vdash_\eta Q :: (d : \beta)$. `cut` on this and $\mathcal{D}_1$ gives the result.

– $\mathcal{D}$ is by $\mathbf{1}$L, $\otimes$L, $\oplus$L, $\multimap$ L, or &L. Follows the same structure as the case for `cut`.

– $\mathcal{D}$ is by $\mathbf{1}$R, $\otimes$R, $\oplus$R, $\multimap$ R, or &R. The rule must be applied on some type $A \in \alpha$. We have $\mathcal{E}' : A \Rightarrow \beta$ by inversion on $\mathcal{E}$. Since β `structural`, another inversion on $\mathcal{E}'$ gives the necessary subtyping relation(s). We match these with the sub-derivations from $\mathcal{D}$ which lets us apply the induction hypotheses. The result follows from the same rule using these.

– $\mathcal{D}$ is by μ. Follows immediately from the induction hypothesis.

– $\mathcal{D}$ is by `var`. Follows from the transitivity of $\Rightarrow$.

□

Theorem 5.11 (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. We proceed by induction on the typing derivation.

Case `id` : We use `id` and the fact that $\Rightarrow$ is reflexive.

Case `cut` : Follows from `cut` and the two induction hypotheses. We use the type from the derivation of the first branch to annotate P.

Case $\mathbf{1}$R, $\otimes$R, $\oplus$R, $\multimap$ R, &R : Follows immediately from the corresponding rule and the induction hypotheses.

Case $\mathbf{1}$L, $\otimes$L, $\oplus$L, $\multimap$ L, &L : Follows immediately from the corresponding rule and the induction hypotheses.

Case SubR, SubL : Follows from the induction hypothesis and lemma 5.10.

Case $\sqcap$R : The induction hypotheses give the two premises needed to apply the corresponding rule in the algorithmic system. The only problem is, the annotated processes returned by the two hypotheses need not have the same annotations. We believe taking the intersection of corresponding annotations should be sufficient, though we could not come up with a proof.

Case $\sqcup\mathtt{L}$: Similar to the previous case.

Case $\mu, \mathtt{var}$: Follows immediately from the induction hypotheses.

□

5.5 Type Safety

In this section, we prove type safety for the algorithmic system. Through the equivalence of the algorithmic system to the declarative system, we also establish type safety for the declarative system.

Note that we did not add new forms of processes, so process configurations and reduction are the same as in section 2.5. We have, however, altered process typing quite a bit. This will require a change in the structure of progress and preservation proofs. We also slightly modify configuration typing to use the new rules:

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

Note that we use singular types (rather than multisets of types) at the top level. The other cases ($\mathtt{config}_0$ and $\mathtt{config}_n$) are the same as before. Even though we use the algorithmic typing judgement in this definition, the results below will hold for declarative typing since these systems are equivalent.

5.5.1 Progress

We need to use a slightly different inversion lemma than we did in section 2.5.2:

Lemma 5.12 (Process Inversion)**.** *If* $\Psi \Vdash_{\eta} P :: (c : A)$*,* $\Psi', c : A \Vdash_{\eta} Q :: (d : \beta)$*, and* $\mathtt{proc}_c(P)$ `poised`*, then the following hold:*

- *If* $Q = \text{wait } d\,;\, Q'$ *then* $P = \text{close } c$.
- *If* $Q = x \leftarrow \text{recv } d\,;\, Q'$ *then* $P = \text{send } c\ (e \leftarrow R_e)\,;\, P'$.
- *If* $Q = \text{send } d\ (e \leftarrow R_e)\,;\, Q'$ *then* $P = x \leftarrow \text{recv } c\,;\, P'$.
- *If* $Q = \text{case } d \text{ of } \{lab_k \rightarrow Q'_k\}_{k \in I}$ *then* $P = c.lab_i\,;\, P'$ *and* $i \in I$.
- *If* $Q = d.lab_i\,;\, Q'$ *then* $P = \text{case } c \text{ of } \{lab_k \rightarrow P'_k\}_{k \in I}$ *and* $i \in I$.

Proof. Let $\mathcal{D} : \Psi \Vdash_\eta P :: (c : A)$ and $\mathcal{E} : \Psi', c : A \Vdash_\eta Q :: (d : \beta)$. Since we are only interested in cases where $\mathtt{proc}_d(Q)$ is communicating along c, we can assume $\mathtt{proc}_d(Q)$ `blocked on` c.[5] We proceed by induction on size Ψ + size A + size Ψ' + size β.

- If Ψ, Ψ', or β contains a non-structural type, we use lemma 5.5 to break it down and apply the induction hypothesis on the derivation we get. In cases where there are two derivations, we pick either.
- If A is a μ, we use lemma 5.5 on $\mathcal{D}$ and on $\mathcal{E}$, and apply the induction hypothesis.
- Otherwise, we have Ψ `structural`, Ψ' `structural`, and β `structural`, and we know A is not a μ.
  - If $A = A_1 \sqcap A_2$, corollary 5.7 on $\mathcal{E}$ gives either $\mathcal{E}_1 : \Psi', c : A_1 \Vdash_\eta Q :: (d : \beta)$ or $\mathcal{E} : \Psi', c : A_2 \Vdash_\eta Q :: (d : \beta)$. In the former case, inversion on $\mathcal{D}$ gives $\mathcal{D}_1 : \Psi \Vdash_\eta P :: (c : A_1)$ and we can immediately apply the induction hypothesis on $\mathcal{D}_1$ and $\mathcal{E}_1$. The latter case is symmetric.
  - If $A = A_1 \sqcup A_2$, then we apply corollary 5.7 on $\mathcal{D}$ this time and use inversion on $\mathcal{E}$. We get matching cases and apply the induction hypothesis.
  - Otherwise, A must be a structural type, $\mathcal{D}$ must be by a structural rule on the right, and $\mathcal{E}$ must be by a structural rule on c. Inversion on $\mathcal{E}$ and the form of Q forces the form of A (e.g. if $Q = \text{wait } d\,;\, Q'$ then $A = 1$). Then, inversion on $\mathcal{D}$ reveals the form of P, and the form of A forces it to be complementary to the form of Q.

□

Given lemma 5.12, progress follows quite trivially:

Theorem 5.13 (Progress)**.** *If* $\models \Omega :: \Psi$ *then either*

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

Proof. We proceed by induction on the derivation of $\models \Omega :: \Psi$. The case for multiple channels follows immediately from the induction hypotheses. For the single channel case, we know $\Omega = \Omega', \mathtt{proc}_c(P)$. By inversion, $\mathcal{D} : \Psi_c \Vdash_\emptyset P :: (c : A)$ and $\mathcal{E} :\models \Omega' :: \Psi_c$. By the induction hypothesis, either Ω' takes a step, in which case Ω takes a step and we are done, or Ω' is poised. Assume Ω' is poised. We case on the structure of P:

[5] Otherwise, inversion on Q shows that all of the equalities in the statement of the lemma are false. Thus, the result follows vacuously.

- If P is a forward, a spawn, or a recursive process then Ω steps by `id`, `cut`, or `rec`, respectively.
- If $\texttt{proc}_c(P)$ `poised`, then Ω is poised.
- Otherwise, we must have $\texttt{proc}_c(P)$ `blocked on` d where $c \neq d$. By lemma 5.8, $d : B \in \Psi_c$. Inversion on $\mathcal{E}$ gives $\Psi_d \Vdash_\eta Q :: (d : B)$ where $\texttt{proc}_d(Q) \in \Omega'$. Since Ω' is poised, $\texttt{proc}_d(Q)$ is poised. We can then apply lemma 5.12, which shows that P and Q have matching forms and can take a step together.

□

5.6 Type Preservation

First, we need the following inversion results.

Lemma 5.14 (Inversion of Id)**.** *If* $\Psi \Vdash_\eta c \leftarrow d :: (c : \alpha)$*, then* $\Psi = d : \beta$ *and* $\beta \Rightarrow \alpha$*.*

Proof. By induction on the typing derivation. The case for `id` is immediate. Otherwise, the last rule must be one of $\sqcap\texttt{R}$, $\sqcap\texttt{L}$, $\sqcup\texttt{R}$, $\sqcup\texttt{L}$, $\mu\texttt{R}$, or $\mu\texttt{L}$, in which case we immediately apply the induction hypotheses and combine the results with the corresponding subtyping rule. □

Lemma 5.15 (Inversion of Cut)**.** *If* $\Psi \Vdash_\eta d : A \leftarrow P_d \,;\, Q_d :: (c : \alpha)$ *then there exist* Ψ_1 *and* Ψ_2 *such that* $\Psi = (\Psi_1, \Psi_2)$*,* $\Psi_1 \Vdash_\eta Q_d :: (d : A)$*, and* $\Psi_2, d : A \Vdash_\eta P_d :: (c : \alpha)$*.*

Proof. Let $P = d : A \leftarrow Q_d \,;\, P'_d$ and $\mathcal{D} : \Psi \Vdash_\eta P :: (c : \alpha)$. We proceed by induction on $\mathcal{D}$.

- If $\mathcal{D}$ is by `cut`, we have the desired result.
- If $\mathcal{D}$ is by $\sqcap\texttt{L}$ or $\mu\texttt{L}$ on channel $e \in \operatorname{dom}\Psi$, we apply the induction hypothesis to get two derivations. The results follows from the same rule on one of the derivations (depending on whether $e \in \operatorname{dom}\Psi_1$ or $e \in \operatorname{dom}\Psi_2$).
- If $\mathcal{D}$ is by $\sqcup\texttt{R}$ or $\mu\texttt{R}$, the result follows from the induction hypothesis and the same rule on the typing derivation for P'_d.
- Otherwise, $\mathcal{D}$ is by $\sqcap\texttt{R}$ or $\sqcup\texttt{L}$. Since the case for $\sqcup\texttt{L}$ is slightly more complicated, we will show that. We have:

$$\mathcal{D} = \frac{\mathcal{D}_1 : \Psi', e : (\beta, B) \Vdash_\eta P :: (c : \alpha) \quad \mathcal{D}_2 : \Psi', e : (\beta, C) \Vdash_\eta P :: (c : \alpha)}{\Psi', e : (\beta, B \sqcup C) \Vdash_\eta P :: (c : \alpha)}\ \sqcup\texttt{L}$$

Induction hypothesis on $\mathcal{D}_1$ gives Ψ_1^B, Ψ_2^B such that $\Psi', e : (\beta, B) = (\Psi_1^B, \Psi_2^B)$, $\mathcal{E}_1 : \Psi_1^B \Vdash_\eta Q_d :: (d : A)$ and $\mathcal{E}_2 : \Psi_2^B, d : A \Vdash_\eta P'_d :: (c : \alpha)$. Similarly, from $\mathcal{D}_2$ we get Ψ_1^C, Ψ_2^C such that $\Psi', e : (\beta, C) = (\Psi_1^C, \Psi_2^C)$, $\mathcal{F}_1 : \Psi_1^C \Vdash_\eta Q_d :: (d : A)$ and $\mathcal{F}_2 : \Psi_2^C, d : A \Vdash_\eta P'_d :: (c : \alpha)$. By corollary 4.11, we know $\operatorname{dom} \Psi_1^B = \operatorname{dom} \Psi_1^C$ and $\operatorname{dom} \Psi_2^B = \operatorname{dom} \Psi_2^C$, so we get the desired results by an application of $\sqcup\mathtt{L}$ on $\mathcal{E}_1$ and $\mathcal{F}_1$ or $\mathcal{E}_2$ and $\mathcal{F}_2$, depending on whether $e \in \Psi_1^B$ or $e \in \Psi_2^B$.

□

Proof of preservation is a lot more cumbersome, as we have to roll a different induction for each case.

Theorem 5.16 (Preservation)**.** *If* $\models \Omega :: \Psi$ *and* $\Omega \longrightarrow \Omega'$ *then* $\models \Omega' :: \Psi$.

Proof. By lemma 2.7, it suffices to consider the subtree which types the root of reduction. So, assume $\Omega_1 \longrightarrow \Omega_2$ and $\models \Omega_1 :: (c : A)$ where c is the root of $\Omega_1 \longrightarrow \Omega_2$. We need to show $\models \Omega_2 :: (c : A)$.

By inversion, $\Omega_1 = (\Omega_c, \mathtt{proc}_c(P))$, $\mathcal{D} : \Psi_c \Vdash_\emptyset P :: (c : A)$, and $\mathcal{E} : \models \Omega_c :: \Psi_c$. We proceed by case analysis on $\Omega_1 \longrightarrow \Omega_2$.

`id :` $P = c \leftarrow d$ and $\Omega_2 = [c/d]\Omega_c$. By lemma 5.14, $\Psi_c = d : D$ where $D \Rightarrow A$. From lemma 5.10 and $\mathcal{E}$, we know $\models \Omega_c :: (d : A)$, thus, $\models [c/d]\Omega_c :: (c : A)$.

`cut :` $P = x : A \leftarrow Q_x\,;\, P'_x$ and $\Omega_2 = \Omega_c, \mathtt{proc}_a(Q_a), \mathtt{proc}_c(P'_a)$ where a is fresh. The result follows straightforwardly from lemma 5.15 and substitution.

`rec :` Follows by a standard substitution lemma for process variables.

In all other cases, we have $\mathtt{proc}_c(P)$ `blocked on` d. By lemma 5.8, $\Psi_c = \Psi'_c, d : D$ for some D, and $\mathtt{proc}_d(Q) \in \Omega_c$ for some Q. Inversion on $\mathcal{E}$ gives $\Omega_c = \Omega'_c, \Omega_d, \mathtt{proc}_d(Q)$, $\mathcal{F}_1 : \Psi_d \Vdash_\emptyset Q :: (d : D)$ and $\mathcal{F}_2 : \models \Omega_d :: \Psi_d$. We only give a representative subset of all cases:

`one :` $P =$ wait $d\,;\, P'$ and $Q =$ close d. First, we claim $\Psi_d = \emptyset$.

Proof. By induction on $\mathcal{F}_1$. Cases for property rules follow immediately from the induction hypothesis (by picking either branch for $\sqcap\mathtt{R}$ and $\sqcup\mathtt{L}$). The only other valid case is $\mathbf{1}\mathtt{R}$, which requires the context to be empty, as desired. □

This shows $\Omega_d = \emptyset$. Next, we show $\Psi'_c \Vdash_\emptyset P' :: (c : A)$.

Proof. We would like to use induction on $\mathcal{D}$, however, we need to generalize the induction hypothesis. So instead, we show: for all Ψ, β, and α, if $\mathcal{D}' :: \Psi, d : \beta \Vdash_\eta \mathtt{wait}\ d; P' :: (c : \alpha)$, then $\Psi \Vdash_\eta P' :: (c : \alpha)$. Using this with $\mathcal{D}$ gives the desired result.

If $\mathcal{D}'$ is by a property rule on d, we immediately apply the induction hypothesis (either one for $\sqcup\mathtt{L}$). If it is by a property rule on Ψ or A, the result follows from the same rule on the induction hypotheses. Otherwise, $\mathcal{D}'$ must be by $\mathbf{1}\mathtt{L}$, which gives the result immediately. □

At this point, we know $\Omega_2 = \Omega'_c, \mathtt{proc}_c(P')$, which can easily be typed using the previous result.

`tensor` : $P = x \leftarrow \mathtt{recv}\ d; P'_x$ and $Q = \mathtt{send}\ d\ (x \leftarrow R_x); Q'$. In addition, $\Omega_2 = \Omega_c, \Omega_d, \mathtt{proc}_a(R_a), \mathtt{proc}_d(Q'), \mathtt{proc}_c(P'_a)$ where a is fresh. We claim there exist Ψ^1_d, Ψ^2_d, D_1, and D_2 such that $\Psi_d = (\Psi^1_d, \Psi^2_d)$, $\Psi^1_d \Vdash_\emptyset R_a :: (a : D_1)$, $\Psi^2_d \Vdash_\emptyset Q' :: (d : D_2)$, and $\Psi'_c, a : D_1, d : D_2 \Vdash_\emptyset P'_a :: (c : A)$.

Proof. By induction on size Ψ'_c + size A + size Ψ_d + size D and using $\mathcal{D}$ and $\mathcal{F}_1$. We assume Ψ'_c, A, D, and Ψ_d are sufficiently generalized (all of them are made free, and all but D is made into multisets). We use α instead of A in the remainder of the proof to emphasize it is a multiset of types.

- If Ψ'_c or α contains a recursive type, or Ψ'_c contains an intersection, or α contains a union, we use lemma 5.5 on $\mathcal{D}$ and apply the induction hypothesis. The result follows from $\mu\mathtt{L}$, $\mu\mathtt{R}$, $\sqcap\mathtt{L}$, or $\sqcup\mathtt{R}$, respectively.
- If α contains an intersection, we use lemma 5.5 on $\mathcal{D}$ and apply the induction hypotheses, which give:

$$\mathcal{G}_1 : \Psi^1_d \Vdash_\emptyset R_a :: (a : D^1_1) \qquad \mathcal{H}_1 : \Psi^2_d \Vdash_\emptyset Q' :: (d : D^1_2)$$
$$\mathcal{D}_1 : \Psi'_c, a : D^1_1, d : D^1_2 \Vdash_\emptyset P'_a :: (c : A_1, \alpha')$$
$$\mathcal{G}_2 : \Psi^3_d \Vdash_\emptyset R_a :: (a : D^2_1) \qquad \mathcal{H}_2 : \Psi^4_d \Vdash_\emptyset Q' :: (d : D^2_2)$$
$$\mathcal{D}_2 : \Psi'_c, a : D^2_1, d : D^2_2 \Vdash_\emptyset P'_a :: (c : A_2, \alpha')$$

  By corollary 4.11, $\Psi^1_d = \Psi^3_d$ and $\Psi^2_d = \Psi^4_d$. From lemma 5.4 and $\sqcap\mathtt{L}$ on $\mathcal{D}_1$ and $\mathcal{D}_2$, we have:

$$\mathcal{D}'_1 : \Psi'_c, a : D^1_1 \sqcap D^2_1, d : D^1_2 \sqcap D^2_2 \Vdash_\emptyset P'_a :: (c : A_1, \alpha')$$
$$\mathcal{D}'_2 : \Psi'_c, a : D^1_1 \sqcap D^2_1, d : D^1_2 \sqcap D^2_2 \Vdash_\emptyset P'_a :: (c : A_2, \alpha')$$

Thus, by an application of $\sqcap\mathtt{R}$ on $\mathcal{D}'_1$ and $\mathcal{D}'_2$, we get

$$\Psi'_c, a : D_1^1 \sqcap D_1^2, d : D_2^1 \sqcap D_2^2 \Vdash_\emptyset P'_a :: (c : \alpha).$$

Moreover, applying $\sqcap\mathtt{R}$ on $\mathcal{G}_1$ and $\mathcal{G}_2$, and on $\mathcal{H}_1$ and $\mathcal{H}_2$ gives

$$\Psi_d^1 \Vdash_\emptyset R_a :: (a : D_1^1 \sqcap D_1^2) \qquad \Psi_d^2 \Vdash_\emptyset Q' :: (d : D_2^1 \sqcap D_2^2)$$

respectively. Picking $D_1 = D_1^1 \sqcap D_1^2$ and $D_2 = D_2^1 \sqcap D_2^2$ gives the result.

- The case where Ψ'_c contains a union is similar to the previous one.
- If some $e \in \operatorname{dom} \Psi_d$ contains a recursive type or an intersection, we use lemma 5.5 on $\mathcal{F}_1$ and apply the induction hypothesis. The result follows from $\mu\mathtt{L}$ or $\sqcap\mathtt{L}$, respectively, applied on either Ψ_d^1 or Ψ_d^2, depending on whichever contains e.
- If some $e \in \operatorname{dom} \Psi_d$ contains a union, we use lemma 5.5 on $\mathcal{F}_1$ and apply the induction hypotheses, which give:

$$\mathcal{G}_1 : \Psi_d^1, e : (\beta', B_1) \Vdash_\emptyset R_a :: (a : D_1^1) \qquad \mathcal{H}_1 : \Psi_d^2 \Vdash_\emptyset Q' :: (d : D_2^1)$$

$$\mathcal{D}_1 : \Psi'_c, a : D_1^1, d : D_2^1 \Vdash_\emptyset P'_a :: (c : \alpha)$$

$$\mathcal{G}_2 : \Psi_d^1, e : (\beta', B_2) \Vdash_\emptyset R_a :: (a : D_1^2) \qquad \mathcal{H}_2 : \Psi_d^2 \Vdash_\emptyset Q' :: (d : D_2^2)$$

$$\mathcal{D}_2 : \Psi'_c, a : D_1^2, d : D_2^2 \Vdash_\emptyset P'_a :: (c : \alpha)$$

Here, we assume without loss of generality that $e \in \operatorname{dom} \Psi_d^1$. We have also unified the contexts using corollary 4.11. From lemma 5.4 and $\sqcup\mathtt{R}$ on $\mathcal{G}_1$ and $\mathcal{G}_2$, followed by $\sqcup\mathtt{L}$ on the results, we get:

$$\mathcal{G} : \Psi_d^1, e : \beta \Vdash_\emptyset R_a :: (a : D_1^1 \sqcup D_1^2).$$

From $\sqcap\mathtt{R}$ on $\mathcal{H}_1$ and $\mathcal{H}_2$, we get

$$\mathcal{H} : \Psi_d^2 \Vdash_\emptyset Q' :: (d : D_2^1 \sqcap D_2^2).$$

Finally, by lemma 5.4 and $\sqcap\mathtt{L}$ on $\mathcal{D}_1$ and $\mathcal{D}_2$, followed by $\sqcup\mathtt{L}$ on the results, we have

$$\Psi'_c, a : D_1^1 \sqcup D_1^2, d : D_2^1 \sqcap D_2^2 \Vdash_\emptyset P'_a :: (c : \alpha).$$

We pick $D_1 = D_1^1 \sqcup D_1^2$ and $D_2 = D_2^1 \sqcap D_2^2$ for the result.

- Otherwise, we have Ψ'_c `structural`, Ψ_d `structural`, and α `structural`, which means corollary 5.7 becomes applicable on $\mathcal{D}$ and $\mathcal{F}_1$. We case on the structure of D:

- If D is a recursive type, $\mathcal{D}$ must be by $\mu\mathtt{L}$ and $\mathcal{F}_1$ must be by $\mu\mathtt{R}$. We apply the induction hypotheses on the premises, which immediately gives the result.
- If $D = D_1 \sqcap D_2$, then inversion on $\mathcal{F}_1$ gives $\mathcal{G}_1 : \Psi_d \Vdash_\emptyset Q :: (d : D_1)$ and $\mathcal{G}_2 : \Psi_d \Vdash_\emptyset Q :: (d : D_2)$. Additionally, Corollary 5.7 on $\mathcal{D}$ gives $\mathcal{H}_1 : \Psi_c, d : D_1 \Vdash_\emptyset P :: (c : \alpha)$ or $\mathcal{H}_2 : \Psi_c, d : D_2 \Vdash_\emptyset P :: (c : \alpha)$. Applying induction hypothesis with $\mathcal{G}_1$ and $\mathcal{H}_1$ or with $\mathcal{G}_2$ and $\mathcal{H}_2$ (depending on which side of the or we have) gives the result.
- If $D = D_1 \sqcup D_2$, we apply inversion on $\mathcal{D}$ and corollary 5.7 on $\mathcal{F}_1$. We get matching cases, and we can apply the induction hypothesis.
- Otherwise, D `structural`. Inversion on $\mathcal{F}_1$ gives $D = D_1 \otimes D_2$ and the corresponding typing derivations for R_x and Q'. Inversion on $\mathcal{D}$ gives the typing derivation for P'_x. We get the desired result by substituting a for x.

□

Finally, we can split Ω_d into Ω_d^1 and Ω_d^2 such that $\models \Omega_d^1 :: \Psi_d^1$ and $\models \Omega_d^2 :: \Psi_d^2$. It is simple to put together a derivation for Ω_2 using the results above.

`internal`, `lolli`, `external` **:** Similar to above.

□

Chapter 6

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 simplifying the presentation or are left as future work. For example, integrating an underlying functional language [22], adding shared channels [4, 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 [3] and abstract types. Their interaction with subtyping, intersections, and unions is an interesting avenue for future work.

Appendix A

Concrete Syntax

The formal grammar of the concrete syntax is presented below. We use `[`*pattern*`]` for optional productions and `{`*pattern*`}` for zero or more repetitions.

⟨*top*⟩ ::= `{` ⟨*typedef*⟩ | ⟨*typesig*⟩ | ⟨*procdef*⟩ `}`

⟨*typedef*⟩ ::= '`type`' ⟨*typename*⟩ '`=`' ⟨*type*⟩

⟨*typesig*⟩ ::= ⟨*procname*⟩ '`:`' ⟨*type*⟩

⟨*procdef*⟩ ::= ⟨*channel*⟩ '`<-`' ⟨*procname*⟩ ⟨*args*⟩ '`=`' ⟨*proc*⟩

⟨*type*⟩ ::= ⟨*typename*⟩
| '`1`'
| ⟨*type*⟩ '`*`' ⟨*type*⟩
| '`+`' '`{`' ⟨*fields*⟩ '`}`'
| ⟨*type*⟩ '`-o`' ⟨*type*⟩
| '`&`' '`{`' ⟨*fields*⟩ '`}`'
| ⟨*type*⟩ '`and`' ⟨*type*⟩
| ⟨*type*⟩ '`or`' ⟨*type*⟩
| '`(`' ⟨*type*⟩ '`)`'

⟨*proc*⟩ ::= ⟨*channel*⟩ '`<-`' ⟨*channel*⟩
| ⟨*channel*⟩ '`<-`' ⟨*procname*⟩ ⟨*args*⟩ '`;`' ⟨*proc*⟩
| '`close`' ⟨*channel*⟩
| '`wait`' ⟨*channel*⟩ '`;`' ⟨*proc*⟩
| '`send`' ⟨*channel*⟩ '`(`' ⟨*channel*⟩ '`<-`' ⟨*proc*⟩ '`)`' '`;`' ⟨*proc*⟩
| '`send`' ⟨*channel*⟩ ⟨*channel*⟩ '`;`' ⟨*proc*⟩
| ⟨*channel*⟩ '`<-`' '`recv`' ⟨*channel*⟩ '`;`' ⟨*proc*⟩

| ⟨*channel*⟩ '`.`' ⟨*label*⟩ '`;`' ⟨*proc*⟩
| '`case`' ⟨*channel*⟩ '`of`' `{` ⟨*branch*⟩ `}`

⟨*args*⟩ ::= `{` ⟨*channel*⟩ `}`

⟨*fields*⟩ ::= ⟨*field*⟩ '`,`' ... '`,`' ⟨*field*⟩

⟨*field*⟩ ::= ⟨*label*⟩ '`:`' ⟨*type*⟩

⟨*branch*⟩ ::= ⟨*label*⟩ '`->`' ⟨*proc*⟩

Here, ⟨*typename*⟩, ⟨*procname*⟩, ⟨*channel*⟩, and ⟨*label*⟩ consist of alphanumerical characters, except ⟨*channel*⟩ starts with a backtick (‘). ⟨*typename*⟩ starts with an uppercase letter, whereas ⟨*procname*⟩ and ⟨*label*⟩ both start with lowercase letters.

Appendix B

Another Example: Bit Strings

Here, we give a slightly more involved example where we define a more interesting property using recursive refinements.

First, we define process level bit string:

```
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].

Bibliography

[1] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. In *POPL*, pages 104–118. ACM Press, 1991.

[2] Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. Intersection and union types: Syntax and semantics. *Inf. Comput.*, 119(2):202–230, 1995.

[3] Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. Behavioral polymorphism and parametricity in session-based communication. In *ESOP*, volume 7792 of *Lecture Notes in Computer Science*, pages 330–349. Springer, 2013.

[4] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, volume 6269 of *Lecture Notes in Computer Science*, pages 222–236. Springer, 2010.

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

[6] Flemming Damm. Subtyping with union types, intersection types and recursive types II. Research Report RR-2259, INRIA, 1994.

[7] Flemming M. Damm. Subtyping with union types, intersection types and recursive types. In *TACS*, volume 789 of *Lecture Notes in Computer Science*, pages 687–706. Springer, 1994.

[8] Henry DeYoung, Luís Caires, Frank Pfenning, and Bernardo Toninho. Cut reduction in linear logic as asynchronous session-typed communication. In *CSL*, volume 16 of *LIPIcs*, pages 228–242. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[9] Joshua Dunfield and Frank Pfenning. Type assignment for intersections and unions in call-by-value languages. In *FoSSaCS*, volume 2620 of *Lecture Notes in Computer Science*, pages 250–266. Springer, 2003.

[10] Joshua Dunfield and Frank Pfenning. Tridirectional typechecking. In *POPL*, pages 281–292. ACM, 2004.

[11] Tim Freeman and Frank Pfenning. Refinement types for ML. In *PLDI*, pages 268–277. ACM, 1991.

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

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

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

[15] Dennis Griffith. *Polarized Substructural Session Types*. PhD thesis, University of Illinois at Urbana-Champaign, April 2016. In preparation.

[16] Kohei Honda. Types for dyadic interaction. In *CONCUR*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993.

[17] Zhaohui Luo, Sergei Soloviev, and Tao Xue. Coercive subtyping: Theory and implementation. *Inf. Comput.*, 223:18–42, 2013.

[18] Frank Pfenning and Dennis Griffith. Polarized substructural session types. In *FoSSaCS*, volume 9034 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2015.

[19] John C. Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon University, June 1996.

[20] Robert J. Simmons. *Substructural Logical Specifications*. PhD thesis, Carnegie Mellon University, 2012.

[21] Christopher A. Stone and Andrew P. Schoonmaker. Equational theories with recursive types. Unpublished Manuscript, 2005.

[22] Bernardo Toninho, Luís Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In *ESOP*, volume 7792 of *Lecture Notes in Computer Science*, pages 350–369. Springer, 2013.

[23] Bernardo Toninho, Luís Caires, and Frank Pfenning. Corecursion and non-divergence in session-typed processes. In *TGC*, volume 8902 of *Lecture Notes in Computer Science*, pages 159–175. Springer, 2014.

[24] Jerome Vouillon and Paul-André Melliès. Semantic types: a fresh look at the ideal model for types. In *POPL*, pages 52–63. ACM, 2004.