Exceptions in a Message Passing Interpretation of Substructural Logic

Shengchao Yang

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Keywords: Exceptions, Affine Logic, Session Types, Concurrent communication

Abstract

Session types are used to describe the structure of communications across channels. Previous research has established a message-passing interpretation of intuitionistic linear logic. Meanwhile, communication failures have been an important research topic in session types. However, the exception handling mechanism has not been well studied in the context of message passing. To bridge this gap, we studied the interpretation of classical affine logic and proposed a new type system containing features such as explicit channel cancellation and exception handling constructors. Our type system ensures program correctness by enforcing session fidelity and deadlock freedom. To experiment, we implemented an interpreter for our language and tested it on several examples to match the expected process behavior. Additionally, we explore the possibility of representing some programming features, such as non-exhaustive matches, in our language using the exception mechanism.

Acknowledgements

TBD

Contents

Chapter 1

Introduction

Linear logic is a refinement of intuitionistic logic that does not satisfy weakening and contraction. It emphasizes the management of resources, where each formula should be used exactly once. Informally speaking, the assumption cannot be too strong for the conclusion, such as $A, B \vdash B$. The Curry-Howard isomorphism states a correspondence between logical proof theory and computational type theory [How80]. The computational interpretation of linear logic, firstly given by Abramsky, lays the foundation for the connection between π-calculus and linear logic [Abr93]. Later on, an expressive formulation of intuitionistic linear logic provides a correspondence between linear proofs and processes, which gives rise to two important ideas: proofs as processes and cut as computation [CPT14].

On the other hand, exceptions have been a practical topic during the development of programming languages. Programmers can utilize exceptions to write explicit control flow for their code. In most functional programming languages, exceptions are managed by expression constructors. For instance, in Standard ML, exceptions are raised by the keyword `raise` and caught by the keyword `handle` [Mil+97]. In concurrent models, communication failures remain both inevitable and critical. Previous research applies affine session types that relax the linearity of session types to incorporate error handling [MV14].

Given the relationship between linear propositions and session types, our goal is to implement exceptions under the computational interpretation of linear logic. However, exceptions are inherently incompatible with linear logic. Let us assume we have some kind of exception handler where part of its processes are designated to deal with exceptional situations. In other words, it will be activated if an exception is raised. This violates linearity, as the resources allocated for the exception handler will not be utilized if no exceptions occur. Affinity, on the other hand, allows us to model such behavior, where resources that are not used can be weakened.

Our solution works under affine systems due to the nature of exceptions explained above. Weakening provides a way to discard resources that are not used, and we interpret this structural rule as a process `cancel` to cancel a channel. We add two new constructors to the type system: `raise` and `try catch`. Although these constructors look similar to most exception handling mechanisms in programming languages, they are interpreted as processes, where the first signals an exception during the computation of processes, and the second catches the signal and activates an additional process that handles the exception concurrently.

We briefly summarize the outline and contributions of this thesis. In the rest of Chapter 1, we will introduce the correlation between sequent proofs and processes, classical affine logic, and exceptions in functional programming languages. Chapter 2 will present the new type system we propose that deals with exceptions and its statics. Computation steps and meta theories that ensures safety of our type system will be formalized and proved in Chapter 3. In Chapter 4, we will demonstrate the implementation of the language interpreter and provide examples of traced processes. Finally, Chapter 5 will show how to represent non-exhaustive matches under our type system, with commentary on related work, limitations of the current approach, and avenues for future research.

1.1 Proofs as processes

A sequent proof of a proposition provides its computational meaning. We use proof terms to record the structure of proofs such that we can reconstruct the proof by its proof term. In the context of intuitionistic linear logic, the proof term corresponds to a process. Below is an annotated sequent proof in intuitionistic linear logic.

$$x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n \vdash P :: c : C$$

where variables $x_1, x_2, \ldots, x_n, c$ represents distinct channels, propositions $A_1, A_2, \ldots, C$ represents session types, and annotation $x_i : A_i$ means the messages send along channel x_i must obey the communication behavior specified by A_i. Under such setting, we can regard the sequent as a process P that uses all the channels in the antecedent and provides a channel in the succedent.

In proof theory, the cut allows for a composition of two separate proofs into a single proof, and correspondingly, a parallel composition of two processes connected by a channel.

$$\frac{\Gamma \vdash P :: x : A \quad \Gamma', x : A \vdash Q :: c : C}{\Gamma, \Gamma' \vdash P \mid Q :: c : C}\ \mathrm{cut}_A$$

The left premise represents a process P that provides channel x, and the right premise represents another process Q that uses channel x. In other words, the cut process P communicates with process Q through channel x specified by A. The linearity of propositions ensures that the channel x has only two endpoints, and other processes besides P and Q will not have access to channel x, since every channel has a distinct variable name.

The cut rule can also be understood as the spawning of a new process. The main thread spawns a new process P, splits the current resources, creates a new channel x, assigns it as process P's endpoint, and continues to execute the rest of the process Q with the remaining resources in parallel.

1.2 Classical linear logic

In classical logic, the judgement has the following form.

$$A_1, A_2, \dots, A_n \vdash C_1, C_2, \dots, C_m$$

which means the conjunction of antecedents $\bigwedge_{i=1}^{n} A_i$ implies the disjunction of the succedents $\bigvee_{j=1}^{m} C_j$. The negation of a proposition becomes a new primitive connective in classical logic, which distinct from intuitionistic logic where $\neg A$ is as same as $A \supset \bot$.

$$\frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta}\ \text{negL} \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta}\ \text{negR}$$

The behavior of negation flips around the proposition between the antecedent and the succedent. In other words, it assumes the contrary and tries to derive a contradiction.

Previous research has shown that classical linear logic can be viewed as π-calculus by interpreting the negation as a dual operation on session types [Wad12]. Defining the dual of a session type A is $A^{\perp}$, we can rewrite the above the judgement in a single-sided form.

$$\vdash A_1^{\perp}, A_2^{\perp}, \dots, A_n^{\perp}, C_1, C_2, \dots, C_m$$

Instead of cutting a proposition on different sides of inference, we cut a proposition against its dual.

$$\frac{\vdash A, \Delta \quad \vdash A^{\perp}, \Delta'}{\vdash \Delta, \Delta'}\ \text{cut}_A$$

In classical logic, we know a proposition A is either true or its dual proposition $A^{\perp}$ is true. In the above cut rule, we can eliminate the usage of A by connecting the remaining context using disjunction.

Under the interpretation of proofs as processes, we can write the annotated two-sided judgement as follows.

$$x_1 : A_1, x_2 : A_2, \dots, x_n : A_n \vdash P :: y_1 : C_1, y_2 : C_2, \dots, y_m : C_m$$

This judgement can be viewed as a process P that communicates along channels $x_1, x_2, \dots, x_n$ and $y_1, y_2, \dots, y_m$, where each channel obeys its protocol requirements. We do not distinguish whether a channel is used or provided, because we can always use the negation rule to flip the propositions. In other word, classical logic blurs the distinction between client and server roles in concurrent communication. It is possible to demonstrate the safety of this system by converting it back to intuitionistic linear logic through double negation translation [Fri78]. Such translation has practical applications, such as continuation passing style translation.

1.3 Affine logic

Affine logic is a substructural logic that rejects the structural rule of contraction. In other words, it is a form of linear logic that retains the weakening rule. There are two ways of formalizing weakening rules. The first formulation is implicit weakening, which modifies the original rules, such as the identity rule, to allow for the closure of the proof by silently discarding all propositions in the context.

$$\frac{}{\Gamma \vdash \mathbf{1}}\ \mathbf{1}R \qquad \frac{}{\Gamma, A \vdash A}\ \text{identity}$$

Another formulation adds the explicit weakening rules.

$$\frac{\Gamma \vdash C}{\Gamma, A \vdash C}\ \text{weaken}$$

Under the classical setting, we will have two weakening rules, one for the antecedent and another one for the succedent. We can establish the equivalence of these two formulations. However, concerning the computational interpretation, the explicit formulation holds greater favorability for two reasons. Firstly, it offers explicit resource control for programmers. Let us annotate the explicit weakening rule as follows.

$$\frac{\begin{array}{c} P \\ \Gamma \vdash c : C \end{array}}{\Gamma, x : A \vdash c : C}\ \text{weaken}$$

We can interpret this rule as a process that discards the future usage of channel x from resources and proceeds with process P. Discarding resources is a common practice in programming. For example, below is an implementation of a function that returns the length of a list.

```
let rec length : 'a list -> int = function
  | [] -> 0
  | _ :: xs -> 1 + length xs
```

In the inductive case where the list is not empty, the function discards the head of the list and proceeds with the remain list. If we enforce linearity in this function, we will have to do unnecessary operations to use the element of the list, which is not appropriate.

Besides the usage of dropping resources, static checking for the interpreter does not align well with implicit weakening. According to implicit rule, when the process reaches its end, all channels in the context will be discarded. This fails to verify whether the programmer intentionally or accidentally left some resources unused.

Chapter 2

Type system

In this chapter we introduce the static semantic for our system. First we list the syntax and explain the behavior for the propositions (types) we include in the system. Next, we present the process syntax and the inference rules that defines the typing judgement for processes.

2.1 Propositions

The syntax of the propositions are defined in Figure 2.1.

$$
\begin{array}{lcll}
\text{Propositions } A, B & ::= & \mathbf{1} & \text{'One', nullary case of } \otimes \\
 & | & \bot & \text{'Bottom', nullary case of } \parr \\
 & | & A \otimes B & \text{'Tensor', communicates with } A \text{ and behaves as } B \\
 & | & A \parr B & \text{'Par', communicates with } A \text{ and behaves as } B \\
 & | & \&\{l : A_l\}_{l \in L} & \text{'With', } n\text{-ary choices distinguished by labels } l \\
 & | & \oplus\{l : A_l\}_{l \in L} & \text{'Plus', } n\text{-ary choices distinguished by labels } l
\end{array}
$$

Figure 2.1: Syntax of propositions

$$
\begin{array}{rcll}
(A)^{\perp} & \longrightarrow & A^{\perp} & A \text{ atom} \\
\mathbf{1}^{\perp} & \longrightarrow & \bot & \\
\bot^{\perp} & \longrightarrow & \mathbf{1} & \\
(A \otimes B)^{\perp} & \longrightarrow & A^{\perp} \parr B^{\perp} & \\
(A \parr B)^{\perp} & \longrightarrow & A^{\perp} \otimes B^{\perp} & \\
(\oplus\{l : A_l\}_{l \in L})^{\perp} & \longrightarrow & \&\{l : A_l^{\perp}\}_{l \in L} & \\
(\&\{l : A_l\}_{l \in L})^{\perp} & \longrightarrow & \oplus\{l : A_l^{\perp}\}_{l \in L} &
\end{array}
$$

Figure 2.2: Dual operations on propositions

The propositions can be classified into two categories: multiplicatives ($\otimes, \parr$) and additives ($\oplus, \&$). The behavior of each type depends on whether it is in the antecedent or the succedent of a sequent. Because of the dual operation in classical logic mentioned in Figure 2.2, the behavior of a session type in the antecedent is the same as its dual in the succedent and vice versa. It is worth noticing that taking the dual of a proposition twice will result in the original proposition, i.e., $(A^\perp)^\perp = A$.

To simplify the explanation of the propositions, we discuss their logical behavior in the succedent through their inference right rules.

- One

  $$\frac{}{\cdot \vdash \mathbf{1}}\ \mathbf{1}R$$

  The empty truth, called One, holds only if there are no resources. This is the nullary case of the multiplicative conjunction.

- Bottom

  $$\frac{\Gamma \vdash \Delta}{\Gamma \vdash \bot, \Delta}\ \bot R$$

  Bottom is a nullary case of the multiplicative disjunction. This proposition does not contain any resources and can be dropped freely.

- Tensor

  $$\frac{\Gamma_1 \vdash A, \Delta_1 \quad \Gamma_2 \vdash B, \Delta_2}{\Gamma_1, \Gamma_2 \vdash A \otimes B, \Delta_1, \Delta_2}\ \otimes R$$

  The multiplicative conjunction $A \otimes B$ is true if A and B are both true. We need to subdivide the resources, use some of them to prove A the other to prove B.

- Par

  $$\frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \parr B, \Delta}\ \parr R$$

  The multiplicative disjunction $A \parr B$ is true if given a refutation of A, B is true, or given a refutation of B, A is true in the current context.

- With

  $$\frac{\Gamma \vdash A_l, \Delta \quad \forall l \in L}{\Gamma \vdash \&\{l : A_l\}_{l \in L}, \Delta}\ \& R$$

  The additive conjunction $\&\{l : A_l\}_{l \in L}$ is true if for all $l \in L$, A_l are true separately with the current resources.

- Plus

  $$\frac{\Gamma \vdash A_k, \Delta \quad k \in L}{\Gamma \vdash \oplus\{l : A_l\}_{l \in L}, \Delta}\ \oplus R_k$$

The additive disjunction $\oplus\{l : A_l\}_{l \in L}$ is true if there exists some $k \in L$ such that A_k is true with the current resources. Therefore, the number of rules is equal to the number of labels in L.

The linear implication $A \multimap B$ is not included in the system because it can be derived from $A^{\perp} \parr B$. We can prove this equivalence. Assume $\Gamma \vdash A \multimap B$, by the invertibility, we have $\Gamma, A \vdash B$. Using the dual operator, we can derive $\Gamma \vdash A^{\perp}, B$. By $\parr R$, we conclude $\Gamma \vdash A \parr B$. Since the proof is invertible, we can derive the other direction as well.

We include the inference rules of identity and cut in our system.

$$\frac{}{A \vdash A}\ \text{identity}_A \qquad \frac{\Gamma_1 \vdash A, \Delta_1 \quad \Gamma_2, A \vdash \Delta_2}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ \text{cut}_A$$

The identity rule closes the proof by ensuring that there is only one identical proposition on each side of the sequent. The cut rule allows us to split the context to prove some arbitrary propositions A and then combine the results to prove the conclusion with the remaining context.

The structural rule of weakening is included in the system. Since we are working with a two-sided sequent, we have weakening rules for both the antecedent and the succedent.

$$\frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta}\ \text{weaken}L \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash A, \Delta}\ \text{weaken}R$$

To avoid arbitrary splitting of the context and extra premises, we modify the $\otimes R$ as the following:

$$\frac{\Gamma \vdash B, \Delta}{\Gamma, A \vdash A \otimes B, \Delta}\ \otimes R^*$$

To prove this rule sensible in the original logical system, we need to show that the original $\otimes R$ can be derived from $\otimes R^*$ and vice versa.

- Assume we have a derivation $\mathcal{D}_1$ for $\Gamma_1 \vdash A, \Delta_1$ and $\mathcal{D}_2$ for $\Gamma_2 \vdash B, \Delta_2$. We can create a derivation for $\Gamma_1, \Gamma_2 \vdash A \otimes B, \Delta_1, \Delta_2$ without using $\otimes R$ rule as the following:

$$\frac{\overset{\mathcal{D}_1}{\Gamma_1 \vdash A, \Delta_1} \qquad \dfrac{\overset{\mathcal{D}_2}{\Gamma_2 \vdash B, \Delta_2}}{\Gamma_2, A \vdash A \otimes B, \Delta_2}\ \otimes R^*}{\Gamma_1, \Gamma_2 \vdash A \otimes B, \Delta_1, \Delta_2}\ \text{cut}_A$$

- Assume we have a derivation $\mathcal{D}$ for $\Gamma \vdash B, \Delta$. We can create a derivation for $\Gamma, A \vdash A \otimes B, \Delta$ without using $\otimes R^*$ rule as the following:

$$\frac{\dfrac{}{A \vdash A}\ \text{identity}_A \qquad \overset{\mathcal{D}}{\Gamma \vdash B, \Delta}}{\Gamma, A \vdash A \otimes B, \Delta}\ \otimes R$$

We can rewrite $\parr L^*$ rule accordingly and justify its equivalence with the original $\parr L$.

$$\frac{\Gamma \vdash B, \Delta}{\Gamma, A \parr B \vdash A, \Delta}\ \parr L^*$$

2.2 Processes

In this section, we will annotate the proofs with proof terms as interpret them as processes. Moreover, we will introduce our exception handling mechanism in our system. We first present the syntax of the processes in Figure 2.3.

$$\begin{array}{rrcl}
\text{Messages} & M & ::= & () \\
 & & \mid & l \\
 & & \mid & x \\
\text{Continuations} & K & ::= & (() \Rightarrow P) \\
 & & \mid & (l \Rightarrow P_l)_{l \in L} \\
 & & \mid & (x \Rightarrow P(x)) \\
\text{Processes} & P, Q & ::= & \mathbf{send}\ x\ M \\
 & & \mid & \mathbf{recv}\ x\ K \\
 & & \mid & \mathbf{fwd}\ y\ x \\
 & & \mid & x\lightning \\
 & & \mid & x \leftarrow P(x);\ Q(x) \\
 & & \mid & \mathbf{raise}\ M \\
 & & \mid & x \twoheadleftarrow \mathbf{try}\ P(x)\ \mathbf{catch}\ Q(x) \\
 & & \mid & \cdot \\
 & & \mid & P; Q
\end{array}$$

Figure 2.3: Syntax of processes

We separate the terms into three categories: messages, continuations, and processes. Variables are used to represent channels and strings are used to represent labels. We require the channels to have distinct names. The information passed through channels is called messages, which can be either a unit message $()$, a label l, or a channel x. The continuation specifies the behavior of the process after receiving a message. Correspondingly, there are three kinds of continuations. $(() \Rightarrow P)$ is a unit continuation that waits for a unit message and continues with process P. $(l \Rightarrow P_l)_{l \in L}$ is a branching continuation that selects to execute process P_l based on the label l it receives. In other words, it has prepared the same number of processes as the number of labels in L. $(x \Rightarrow P(x))$ is a continuation that waits for a channel x and continues with process $P(x)$. We write $P(x)$ instead of P to indicate that the process P depends on the channel x.

From the description above, we provide an intuition of how processes interact with each other. Imagine a segment where one endpoint outputs a message and passes it to the other endpoint, which works as a continuation that waits for the message through this segment. We can see that each message has its own corresponding continuation. This means that if the message passing through the segment is a label while the continuation is waiting for a channel, the interaction will be stuck and cannot proceed. We can use propositions as session types introduced in Section 2.1 to restrict channel behavior. More details on this observation will be discussed in the operational semantics.

We present the annotated inference rules in Figure 2.4, where proof terms are interpreted as processes. The judgement has the form $\Gamma \vdash P :: \Delta; \Omega$, which means the process P communicates along the channels in Γ and Δ and has an exception handler Ω. Such rules are also called the static semantics of the type system. We first introduce the standard processes and explain how information flows through channels. Processes of **send** and **recv** are elementary processes. **send** $x\ M$ sends a message M along channel x, and **recv** $x\ K$ receives a message from channel x and continues with the continuation K. Whether a process sends or receives a message depends on the polarity of the connective. Positive connectives, whose right rule is not invertible, will send a message in their right rule and receive a message in their left rule. The intuition is that non-invertible rules always make a choice, while invertible rules do not contain any extra information.

The proposition restricts the message type a channel can send or receive. A channel of type $\mathbf{1}$ or $\bot$ can only send or receive a unit message, respectively. In logic systems, $\mathbf{1}$ represents the empty truth, and $\bot$ represents the empty falsehood. Therefore, a unit message is also called an empty message since it does not contain any useful information. An analogy can be drawn from a unit message to computation suspension in functional programming. Under call-by-value settings, an expression e needs to be eagerly evaluated before it can be involved in future computation. However, if we replace e by $\lambda(x : \texttt{unit}.e)$, then the computation of e is suspended since a function is already a value. We can retrieve the computation by applying the function to a unit value $()$. In the message-passing setting, **recv** $x\ (() \Rightarrow P)$ is a "suspended" process that waits for a unit message sent through channel x.

A channel of product type $\otimes$ and $⅋$ can send or receive a channel, and its type evolves accordingly. For instance, in the $[\otimes R^*]$ rule, the process **send** $x\ y; P$ sends a channel y along channel x and continues with process P. Before sending the channel y, the channel x has type $A \otimes B$. After sending y of type A, the channel x evolves to type B. The last message corresponds to choice types $\oplus$ and $\&$. A choice type is a collection of types distinguished by labels. For instance, the type `bool` can be written as $\oplus$true : $\mathbf{1}$, false : $\mathbf{1}$, a choice type of two labels true and false with unit types. **send** $x\ l$ sends a label l along channel x. This can be viewed as a process that makes a choice to select label l from all the possible label collections L. We add a subscript k as a label to the $[\oplus R_k]$ rule to indicate that the process chooses k. The type of the channel evolves from choice type to the corresponding type

$[\mathbf{1}R]$
$$\frac{}{\cdot \vdash \textbf{send}\ x\ () :: x : \mathbf{1}; \Omega}$$

$[\mathbf{1}L]$
$$\frac{\Gamma \vdash P :: \Delta; \Omega}{\Gamma, x : \mathbf{1} \vdash \textbf{recv}\ x\ (() \Rightarrow P) :: \Delta; \Omega}$$

$[\bot R]$
$$\frac{\Gamma \vdash P :: \Delta; \Omega}{\Gamma \vdash \textbf{recv}\ x\ (() \Rightarrow P) :: x : \bot, \Delta; \Omega}$$

$[\bot L]$
$$\frac{}{x : \bot \vdash \textbf{send}\ x\ () :: \cdot; \Omega}$$

$[\otimes R^*]$
$$\frac{\Gamma \vdash P :: x : B, \Delta; \Omega}{\Gamma, y : A \vdash \textbf{send}\ x\ y; P :: x : A \otimes B, \Delta; \Omega}$$

$[\otimes L]$
$$\frac{\Gamma, x : B, y : A \vdash P :: \Delta; \Omega}{\Gamma, x : A \otimes B \vdash \textbf{recv}\ x\ (y \Rightarrow P(y)) :: \Delta; \Omega}$$

$[\parr R]$
$$\frac{\Gamma \vdash P :: \Delta; x : B, y : A, \Omega}{\Gamma \vdash \textbf{recv}\ x\ (y \Rightarrow P(y)) :: x : A \parr B, \Delta; \Omega}$$

$[\parr L^*]$
$$\frac{\Gamma, x : B \vdash P :: \Delta; \Omega}{\Gamma, x : A \parr B \vdash \textbf{send}\ x\ y; P :: y : A, \Delta; \Omega}$$

$[\oplus R_k]$
$$\frac{\Gamma \vdash P :: A_k; \Omega \qquad k \in L}{\Gamma \vdash \textbf{send}\ x\ k; P :: x : \oplus\{l : A_l\}_{l \in L}, \Delta; \Omega}$$

$[\oplus L]$
$$\frac{\Gamma, x : A_l \vdash P_l :: \Delta; \Omega \qquad (\forall l \in L)}{\Gamma, x : \oplus\{l : A_l\}_{l \in L} \vdash \textbf{recv}\ x\ (l \Rightarrow P_l)_{l \in L} :: \Delta; \Omega}$$

$[\& R]$
$$\frac{\Gamma, x : A_l \vdash P_l :: \Delta; \Omega \qquad (\forall l \in L)}{\Gamma, x : \&\{l : A_l\}_{l \in L} \vdash \textbf{recv}\ x\ (l \Rightarrow P_l)_{l \in L} :: \Delta; \Omega}$$

$[\& L_k]$
$$\frac{\Gamma, x : A_k \vdash P :: \Delta; \Omega \qquad k \in L}{\Gamma, x : \&\{l : A_l\}_{l \in L} \vdash \textbf{send}\ x\ k; P :: \Delta; \Omega}$$

$[\textsc{weaken}\mathrm{R}]$
$$\frac{\Gamma \vdash P :: \Delta; \Omega}{\Gamma \vdash x\lightning; P :: x : A, \Delta; \Omega}$$

$[\textsc{weaken}\mathrm{L}]$
$$\frac{\Gamma \vdash P :: \Delta; \Omega}{\Gamma, x : A \vdash x\lightning; P :: \Delta; \Omega}$$

$[\textsc{identity}_A]$
$$\frac{}{y : A \vdash \textbf{fwd}\ x\ y :: x : A; \Omega}$$

$[\textsc{cut}_A]$
$$\frac{\Gamma_1 \vdash P(x) :: x : A, \Delta_1; \cdot \qquad \Gamma_2, x : A \vdash Q(x) :: \Delta_2; \Omega}{\Gamma_1, \Gamma_2 \vdash x \leftarrow P(x);\ Q(x) :: \Delta_1, \Delta_2; \Omega}$$

$[\textsc{empty}]$
$$\frac{}{\cdot \vdash \cdot :: \cdot; \Omega}$$

$[\textsc{raise}]$
$$\frac{\Gamma \vdash P :: \Delta; \cdot}{\Gamma \vdash \textbf{raise}\ P :: \Delta; x : \mathrm{exn}}$$

$[\textsc{trycatch}_A]$
$$\frac{\Gamma_1 \vdash P(y) :: y : A, \Delta_1; x : \mathrm{exn} \qquad \Gamma_2, x : \mathrm{exn}, y : A \vdash \mathrm{erecv}\ x\ Q(y) :: \Delta_2; \Omega}{\Gamma_1, \Gamma_2 \vdash y \twoheadleftarrow \textbf{try}\ P(y)\ \textbf{catch}\ Q(y) :: \Delta_1, \Delta_2; \Omega}$$

Figure 2.4: Static semantics of the type system

of label k. **recv** x $(l \Rightarrow P_l)$ provides a branching continuation that selects one branch to execute based on the label it receives. We write $\forall l \in L$ to indicate that the process has prepared a process for each label in L. In other words, if there is some label missing in the branching, the process will be stuck and cannot proceed. However, we can exploit this feature by including non-exhaustive matches in our language, which will be discussed in the later section.

Now we discuss the structural rules of our system. The weakening rule is represented by the channel cancellation process $x\lightning$. Although we have two different weakening rules: [weakenL] and [weakenR], we do not distinguish their differences in process. The process $x\lightning$ discards the channel x from the context and continues with process P under the remaining context. The [identity_A] rule is represented by the process **fwd** y x, which passes the information from channel y to channel x. The cut rule spawns a child process P with a newly created private channel x and continues with the parent process Q under the remaining context. The split of the exception handling context and the remaining rules about exceptions are discussed in the next section.

2.3 Exception processes

We first examine how exceptions are raised and handled in common functional programming languages. Below is a division function that returns the division of two numbers optionally. In other words, if there is no division by zero, the function returns the division of two numbers. Otherwise, it prints a message to console and returns none.

```
let div (a : int) (b : int) : int option =
  try
    Some (a / b)
  with Failure _ -> print_endline "Division by zero"; None
```

An important observation is sequential execution in computation. Whether the print statement will be executed is not determined until the division is evaluated. We adapt such an idea in our exception handling mechanism. The execution of an exception handling process requires the execution of the main process, and we use Ω to store the channels that communicate between the main process and the exception handling process.

A main process P's Ω is a singleton context, which means it contains at most one channel. If there are no channels in Ω, this means process P does not have any exception handling mechanism. In other words, if P raises an exception, it will be an uncaught exception. Nevertheless, if a process has more than one handling mechanism, we can always combine them into one.

$$\Omega = \cdot \mid x : \text{exn}$$

The channel in Ω has type exn, which is defined as the following:

$$\mathrm{exn} = \oplus\{\mathrm{act} : \mathbf{1}, \mathrm{sil} : \mathbf{1}\}$$

As described before, a channel of type exn sends and receives a label of act or sil. Act label represents activation of the exception handling process when an exception is raised. Conversely, sil label represents the drop of the exception handling process when the process finished executing without any processes.

With enough introduction, we can look at the structure of [TRYCATCH$_A$]. It creates a channel y and splits the resources to execute the main process P. Meanwhile, a new channel x of type `exn` is created and assigned to P. The process `erecv` is a derived form which is defined by the following:

$$\begin{aligned} \Gamma \vdash \mathrm{erecv}\ x\ Q(y) :: \Delta; \Omega \triangleq \Gamma \vdash \mathbf{recv}\ x\ (&\mathrm{act} \Rightarrow \mathbf{recv}\ x\ (() \Rightarrow Q(y)) \\ &\mid \mathrm{sil} \Rightarrow \mathbf{recv}\ x\ (() \Rightarrow y\lightning; \Gamma\lightning; \Delta\lightning)) :: \Delta; \Omega \end{aligned}$$

It waits for a label sent through channel x. If the label is `act`, which means an exception is raised by its main process, it starts to execute the process $Q(y)$ using the remaining context and the original exception handling context. On the other hand, if the main process finishes execution without any exceptions, a `sil` label will lead to a branch of cancelling all resources prepared for exception handling. The exception handling context is inherited from the original process.

Now let's revisit the body of the division function to see how exceptions are raised. The division `a / b` is a partial function that will raise an exception when $b = 0$. The exception is characterized by the exception constructor `Failure` with a message attached to it. In the example above, we do not utilize the information stored in the exception. We follow the same structure in our system; instead of raising an exception with a value, we raise a process as described in [RAISE]. **raise** P raises an exception and continues to execute with process P. Currently, we require the exception to be caught by the exception handling process. This invariant is enforced by requiring Ω to be non-empty. However, this restriction can be relaxed, as uncaught exceptions are common in programming languages. We explore this feature in later chapters.

Chapter 3

Runtime semantics

We record the program as states and the execution of the program as transitions between states. Each state consists of a configuration and a cancelled set. The configuration is a multiset whose reduction follows linear inference, while the cancelled set records the channels that are cancelled during the execution globally. We also formalize recursion to make the program more expressive. After that, we prove the safety theorems of the system by showing that well-typed configurations will not get stuck, and after reduction, the configuration will still be well-typed.

3.1 Configuration

To formalize the runtime semantics, we can view each process as an individual object. The start of the program can be viewed as a configuration with a single object. During the execution of the program, new processes will be spawned, adding new process objects to the configuration. We define the configuration C in Figure 3.1.

$$\begin{array}{llcll}
\text{Configuration} & C & ::= & \cdot & \text{empty configuration} \\
 & & \mid & C_1, C_2 & \text{join configuration} \\
 & & \mid & \mathbf{proc}(\Gamma/\Delta/\Omega/P) & \text{process object}
\end{array}$$

Figure 3.1: Syntax of configuration

Empty configuration is an empty multiset that does not contain any process object. A single process object of form $\mathbf{proc}(\Gamma/\Delta/\Omega/P)$ represents a process P communicates along the channels in Γ and Δ and has an exception handling context Ω. During the transition of the program, the meta variables in the process object may change accordingly. Lastly, the join combines two configuration into one configuration.

To avoid the situation of bad programs, where one channel receives a label message and has a unit continuation, we establish the typing judgement for configuration of form $\Phi \vdash C :: \Phi'$. The configuration C uses channels in Φ and provides channels Φ'. For instance, if a configuration C uses channel $a : A$ and provides channel $b : B$, then $\Phi, a : A \vdash C :: b : B, \Phi$ holds. We provide the relevant rules in Figure 3.2.

$$\textsc{[empty]}\ \frac{}{\Phi \vdash \cdot :: \Phi} \qquad \textsc{[join]}\ \frac{\Phi \vdash C_1 :: \Phi' \qquad \Phi' \vdash C_2 :: \Phi''}{\Phi \vdash C_1, C_2 :: \Phi''}$$

$$\textsc{[object]}\ \frac{}{\Phi, \Gamma \vdash \mathbf{proc}(\Gamma/\Delta/\Omega/P) :: \Delta, \Phi}$$

Figure 3.2: Typing judgement of configuration

[EMPTY] states that an empty configuration does not use any channels and provides the same channels. We add an arbitrary context Φ on both sides to generalize the [OBJECT] rule, since there may be channels that are not used by process P. [JOIN] combines two configurations C_1 and C_2 into one configuration C by some certain order. This may contradict with the intuition that the configuration is an unordered set where C_1, C_2 should be as same as C_2, C_1. Here, we require that there exists an ordering of the configuration such that it is well-typed. Such ordering is not unique, apparently. Consider an example where $C_1 = \mathbf{proc}(\cdot/a : A/\Omega_1/P)$ and $C_2 = \mathbf{proc}(b : B, c : C/\cdot/\Omega_2/Q)$. We can join these two configurations in two ways: $C = C_1, C_2$ and $C = C_2, C_1$ where $b : B, c : C \vdash C :: a : A$. Using the [OBJECT], we can find such $\Phi' = a : A, b : B, c : C$ for the first way and $\Phi' = \cdot$ for the second way. However, if we change C_2 into $\mathbf{proc}(a : A, b : B/\cdot/\Omega_2/Q)$, the overall configuration $b : B \vdash C :: \cdot$ still type checks, but the ordering is unique. It cannot be the case that $C = C_2, C_1$, and the reason for that is because C_1 provides channel a that is used by C_2. By this observation, two configurations can be swapped if none of the channels provided by the first configuration are not used by the second configuration.

Besides the configuration, we also need a globally defined set to store the channels that are cancelled during the execution, call it $\mathcal{S}$. The main reason why we need to keep track of this set is because the cancellation of a channel may affect the execution of other processes. Every channel can be viewed as a segment that has two endpoints. In the original execution, one endpoint will send a message and the other endpoint is a continuation that waits for this message. However, if one endpoint is cancelled, the other endpoint should also be cancelled accordingly. Otherwise, the process will be stuck and cannot proceed. We call this procedure as cancellation propagation.

To conclude, the state of the computation is a pair of current configuration and cancelled sets $\langle C, \mathcal{S} \rangle$.

3.2 Recursion

Recursion is a very practical tool in programming, and we also want such a feature in our system. However, the formalization of recursion has little to do with logics. We achieve such equirecursion by having a signature Σ_{decl} and a structure Σ_{defn}, whose syntax is defined in Figure 3.3.

$$
\begin{array}{rcl}
\Sigma_{\text{decl}} & ::= & \cdot \\
& | & t : A \\
& | & p : (\Gamma, \Delta) \\
& | & \Sigma_{\text{decl1}}, \Sigma_{\text{decl2}} \\
\Sigma_{\text{defn}} & ::= & \cdot \\
& | & t = A \\
& | & p(y_1, y_2, \ldots, y_m)[x_1, x_2, \ldots, x_n] = P \\
& | & \Sigma_{\text{defn1}}, \Sigma_{\text{defn2}}
\end{array}
$$

Figure 3.3: Syntax of signature and structure

A signature is a set of type declarations $t : A$ and process declarations $p : (\Gamma, \Delta)$. Such a signature can be determined when scanning through the whole program. We want to ensure that all the definitions in the program are well-defined based on the declarations. This structure records both type and process definitions. A type definition has the form $t = A$, where t is a type identifier and A is a proposition. A process definition has the form $p(y_1, y_2, \ldots, y_m)[x_1, x_2, \ldots, x_n] = P$. The identification includes a process name p, a list of output channels $y_1, y_2, \ldots, y_m$, and a list of input channels $x_1, x_2, \ldots, x_n$. The process such identification represents is P. We use the judgement $\Sigma_{\text{decl}} \vdash \Sigma_{\text{defn}}$ **struct** to represent that Σ_{defn} is a valid structure given all the declarations in Σ_{decl}. We present the rules for signatures in Figure 3.4. The judgements for well-formed types $\Sigma_{\text{decl}} \vdash A$ **type** and well-formed processes $\Sigma_{\text{decl}} \vdash P$ **proc** can be defined by looking at the signature, and thus we omit the rules here.

$$\textsc{[empty]}\quad \frac{}{\Sigma_{\mathrm{decl}} \vdash \cdot\ \textbf{struct}} \qquad \textsc{[typedef]}\quad \frac{\Sigma_{\mathrm{decl}} \vdash \Sigma_{\mathrm{defn}}\ \textbf{struct} \qquad \Sigma_{\mathrm{decl}} \vdash A\ \textbf{type}}{\Sigma_{\mathrm{decl}} \vdash \Sigma_{\mathrm{defn}}, t = A\ \textbf{struct}}$$

$$\textsc{[procdef]}\quad \frac{\Sigma_{\mathrm{decl}} \vdash \Sigma_{\mathrm{defn}}\ \textbf{struct} \qquad \Sigma_{\mathrm{decl}} \vdash P(x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m)\ \textbf{proc}}{\Sigma_{\mathrm{decl}} \vdash \Sigma_{\mathrm{defn}}, p(y_1, y_2, \ldots, y_m)[x_1, x_2, \ldots, x_n] = P\ \textbf{struct}}$$

Figure 3.4: Judgement of well-formed structure

We present some examples of recursive types that will be used in future examples.

- Natural number `nat`

  $$\texttt{nat} \triangleq \oplus\{\mathrm{zero} : \mathbf{1}, \mathrm{succ} : \texttt{nat}\}$$

  The definition of natural number is a choice type of two labels `zero` and `succ`, where `zero` represents the number 0 and `succ` represents the successor of a natural number. This looks similar to the inductive type $\mu(t.1 + t)$.

- List of natural numbers `list`

  $$\texttt{list} \triangleq \oplus\{\mathrm{nil} : \mathbf{1}, \mathrm{cons} : \texttt{nat} \otimes \texttt{list}\}$$

  List of nature numbers is also a choice type of two labels `nil` and `cons`. In the `cons` label, it contains multiplicative product of `nat` and `list`. To ensure `list` is well-defined, we need to have a type definition for `nat` in the signature.

To utilize the recursion process, we add a new process **call** to activate another process in Figure 3.5. A process can call itself to achieve self recursion.

$$\begin{array}{llcl} \text{Processes} & P, Q & ::= & \ldots \\ & & \mid & \textbf{call}\ p\ (y_1, y_2, \ldots, y_m)\ [x_1, x_2, \ldots, x_n] \end{array}$$

$$\textsc{[call]}\quad \frac{\Sigma_{\mathrm{decl}} \vdash p(y_1, y_2, \ldots, y_m)[x_1, x_2, \ldots, x_n] = P\ \textbf{struct} \qquad \Gamma \vdash P :: \Delta; \Omega}{\Gamma \vdash \textbf{call}\ p\ (y_1, y_2, \ldots, y_m)\ [x_1, x_2, \ldots, x_n] :: \Delta; \Omega}$$

where $\Gamma = x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n$ and $\Delta = y_1 : C_1, y_2 : C_2, \ldots, y_m : C_m$

Figure 3.5: Extended syntax and static semantics of the type system

The **call** process looks up the signature Σ_{decl} and finds the process defined by identifier p. There should be no future processes after the **call**, thus we require process P takes all the channels in Γ and Δ. The exception handling context Ω is inherited from the original process.

3.3 Reduction

In future discussions, we'll distinguish channels allocated using $a, b, c, \ldots$ for newly created channels and use $x, y, z, \ldots$ for channel variables that require substitution by another allocated channel. For instance, $a \leftarrow P(a);\ Q(a)$ should not be allowed since channel a should be a variable instead of an allocated channel. Similarly, $x\lightning$ is invalid since we cannot cancel a channel that is not allocated.

The reduction of the configuration is based on linear inference. New process objects are created when a new process is spawned. Correspondingly, process objects may be destroyed when the process is finished or evolves to another process. A reduction step is represented by $\langle \mathcal{C}; \mathcal{S} \rangle \longrightarrow \langle \mathcal{C}'; \mathcal{S}' \rangle$. The reduction rules only examine parts of the configuration. The untouched parts are preserved in the new configuration. We first present the standard reduction rules without exceptions in Figure 3.6.

Most of the standard rules represent message reduction. For instance, in [LABEL-R], one process sends a label k through channel a, and in another process, it waits for a label from a and passes it to a branching continuation. Such two processes can make progress simultaneously. As the first process continues to execute the remaining process P, the second process chooses process P_k to execute. By the duality of the propositions, each reduction has two versions of rules, one for the right rule and one for the left rule.

In the [LINKING] rule, the process **fwd** $c\ a$ passes information from channel a to channel c. We achieve this effect by replacing all usages of channel a by channel c in the continuation process. Since the invariant that every channel has two endpoints holds, we know there is only one such process where we need to do the substitution. The [SPAWN] rule splits the original object into two process objects. The first object contains the new process P, and the second object contains the original process Q. A new channel a is created and put into the relevant context. We require that the channel a is globally fresh to avoid naming conflicts. Moreover, the contexts are split based on the metavariables in each process. According to the static rules, there must be a valid splitting of the context that maintains the typing of the configuration. In the [CALL] rule, we assume the signature is well-formed, thus there exists a process declaration for p. We find such a definition $p(y_1, y_2, \ldots, y_m)[x_1, x_2, \ldots, x_n] = P$ and execute the process P. We can drop a process object when the process is finished. A process is finished if all the channels in the contexts are used, and there are no remaining processes, which is formalized by the [END] rule.

$$[\textsc{unit-r}]\ \left\langle \begin{array}{l} \mathbf{proc}(\cdot/a:\mathbf{1}/\Omega_1/\mathbf{send}\ a\ ()) \\ \mathbf{proc}(\Gamma, a:\mathbf{1}/\Delta/\Omega_2/\mathbf{recv}\ a\ (() \Rightarrow P)) \end{array} ;\mathcal{S} \right\rangle \longrightarrow \left\langle \begin{array}{l} \mathbf{proc}(\cdot/\cdot/\Omega_1/\cdot) \\ \mathbf{proc}(\Gamma/\Delta/\Omega_2/P) \end{array} ;\mathcal{S} \right\rangle$$

$$[\textsc{unit-l}]\ \left\langle \begin{array}{l} \mathbf{proc}(a:\bot/\cdot/\Omega_1/\mathbf{send}\ a\ ()) \\ \mathbf{proc}(\Gamma/a:\bot, \Delta/\Omega_2/\mathbf{recv}\ a\ (() \Rightarrow P)) \end{array} ;\mathcal{S} \right\rangle \longrightarrow \left\langle \begin{array}{l} \mathbf{proc}(\cdot/\cdot/\Omega_1/\cdot) \\ \mathbf{proc}(\Gamma/\Delta/\Omega_2/P) \end{array} ;\mathcal{S} \right\rangle$$

$$[\textsc{channel-r}]\ \left\langle \begin{array}{l} \mathbf{proc}(\Gamma_1, a:A/c:A\otimes B, \Delta_1/\Omega_1/\mathbf{send}\ c\ a; P) \\ \mathbf{proc}(\Gamma_2, c:A\otimes B/\Delta_2/\Omega_2/\mathbf{recv}\ c\ (x \Rightarrow P(x))) \end{array} ;\mathcal{S} \right\rangle \longrightarrow \left\langle \begin{array}{l} \mathbf{proc}(\Gamma_1/c:B, \Delta_1/\Omega_1/P) \\ \mathbf{proc}(\Gamma_2, c:B/\Delta_2/\Omega_2/P(a \end{array} \right.$$

$$[\textsc{channel-l}]\ \left\langle \begin{array}{l} \mathbf{proc}(\Gamma_1, c:A \parr B/a:A, \Delta_1/\Omega_1/\mathbf{send}\ c\ a; P) \\ \mathbf{proc}(\Gamma_2/c:A \parr B, \Delta_2/\Omega_2/\mathbf{recv}\ c\ (x \Rightarrow P(x))) \end{array} ;\mathcal{S} \right\rangle \longrightarrow \left\langle \begin{array}{l} \mathbf{proc}(\Gamma_1/c:B, \Delta_1/\Omega_1/P) \\ \mathbf{proc}(\Gamma_2, c:B/\Delta_2/\Omega_2/P(a \end{array} \right.$$

$$[\textsc{label-r}]\ \left\langle \begin{array}{l} \mathbf{proc}(\Gamma_1/a:\oplus\{l:A_l\}_{l\in L}, \Delta_1/\Omega_1/\mathbf{send}\ a\ k; P) \\ \mathbf{proc}(\Gamma_2, a:\oplus\{l:A_l\}_{l\in L}/\Delta_2/\Omega_2/\mathbf{recv}\ a\ (l \Rightarrow P_l)_{l\in L}) \end{array} ;\mathcal{S} \right\rangle \longrightarrow \left\langle \begin{array}{l} \mathbf{proc}(\Gamma_1/a:A_k, \Delta_1/\Omega_1/P) \\ \mathbf{proc}(\Gamma_2, a:A_k/\Delta_2/\Omega_2/P_k \end{array} \right.$$

$$[\textsc{label-l}]\ \left\langle \begin{array}{l} \mathbf{proc}(\Gamma_1, a:\&\{l:A_l\}_{l\in L}/\Delta_1/\Omega_1/\mathbf{send}\ a\ k; P) \\ \mathbf{proc}(\Gamma_2/a:\&\{l:A_l\}_{l\in L}, \Delta_2/\Omega_2/\mathbf{recv}\ a\ (l \Rightarrow P_l)_{l\in L}) \end{array} ;\mathcal{S} \right\rangle \longrightarrow \left\langle \begin{array}{l} \mathbf{proc}(\Gamma_1, a:A_k/\Delta_1/\Omega_1/P) \\ \mathbf{proc}(\Gamma_2/a:A_k, \Delta_2/\Omega_2/P_k \end{array} \right.$$

$$[\textsc{linking}]\ \left\langle \begin{array}{l} \mathbf{proc}(a:A/c:A/\Omega_1/\mathbf{fwd}\ c\ a) \\ \mathbf{proc}(\Gamma/a:A, \Delta/\Omega_2/P(a)) \end{array} ;\mathcal{S} \right\rangle \longrightarrow \left\langle \begin{array}{l} \mathbf{proc}(\cdot/\cdot/\Omega_1/\cdot) \\ \mathbf{proc}(\Gamma/c:A, \Delta/\Omega_2/P(c)) \end{array} \right.$$

$$[\textsc{spawn}]\ \left\langle\ \mathbf{proc}(\Gamma_1, \Gamma_2/\Delta_1, \Delta_2/\Omega/x \leftarrow P(x);\ Q(x))\ ;\mathcal{S} \right\rangle \longrightarrow \left\langle \begin{array}{l} \mathbf{proc}(\Gamma_1/a:A, \Delta_1/\cdot/P(a)) \\ \mathbf{proc}(\Gamma_2, a:A/\Delta_2/\Omega/Q(a \end{array} \right.$$

$$[\textsc{call}]\ \left\langle\ \mathbf{proc}(\Gamma/\Delta/\Omega/\mathbf{call}\ p\ (y_1, y_2, \ldots, y_m)\ [x_1, x_2, \ldots, x_n])\ ;\mathcal{S} \right\rangle \longrightarrow \left\langle\ \mathbf{proc}(\Gamma/\Delta/\Omega/P)\ ;\mathcal{S} \right\rangle$$

$$[\textsc{end}]\ \left\langle\ \mathbf{proc}(\cdot/\cdot/\cdot/\cdot)\ ;\mathcal{S} \right\rangle \longrightarrow \left\langle\ \cdot\ ;\mathcal{S} \right\rangle$$

Figure 3.6: Standard reductions of configuration

Now let's examine the reduction rules in channel cancellation as listed in Figure 3.7. The cancellation process removes a channel from the context and adds it to the cancellation set. We have two different rules for right and left cancellation for the channel a in the contexts Γ and Δ, respectively. The channel propagation rule states that if a channel a is cancelled, then the process whose prefix is a should also be cancelled. The prefix of a process is the channel that the process directly communicates with. We list the prefixes of a process below:

$$\begin{aligned} \text{prefix}(\mathbf{send}\ x\ M) &= x \\ \text{prefix}(\mathbf{recv}\ x\ K) &= x \\ \text{prefix}(\mathbf{fwd}\ y\ x) &= y \end{aligned}$$

If the process is not included in the above list, then it does not have a prefix channel. Cancelling a process is achieved by replacing the process with a process that cancels all the channels in the context.

$$
\begin{array}{rlcl}
[\textsc{cancel-r}] & \left\langle\ \mathbf{proc}(\Gamma/a:A,\Delta/\Omega/a\lightning;P)\ \ ;\mathcal{S}\right\rangle & \longrightarrow & \left\langle\ \mathbf{proc}(\Gamma/\Delta/\Omega/P)\ \ ;\mathcal{S}\cup\{a\}\right\rangle \\
[\textsc{cancel-l}] & \left\langle\ \mathbf{proc}(\Gamma,a:A/\Delta/\Omega/a\lightning;P)\ \ ;\mathcal{S}\right\rangle & \longrightarrow & \left\langle\ \mathbf{proc}(\Gamma/\Delta/\Omega/P)\ \ ;\mathcal{S}\cup\{a\}\right\rangle \\
[\textsc{cancel-prop-r}] & \left\langle\begin{array}{l}\mathbf{proc}(\Gamma/a:A,\Delta/\Omega/P)\\ \mathrm{prefix}(P)=a\end{array}\ ;\mathcal{S}\cup\{a\}\right\rangle & \longrightarrow & \left\langle\ \mathbf{proc}(\Gamma/\Delta/\Omega/\Gamma\lightning;\Delta\lightning)\ \ ;\mathcal{S}\cup\{a\}\right\rangle \\
[\textsc{cancel-prop-l}] & \left\langle\begin{array}{l}\mathbf{proc}(\Gamma,a:A/\Delta/\Omega/P)\\ \mathrm{prefix}(P)=a\end{array}\ ;\mathcal{S}\cup\{a\}\right\rangle & \longrightarrow & \left\langle\ \mathbf{proc}(\Gamma/\Delta/\Omega/\Gamma\lightning;\Delta\lightning)\ \ ;\mathcal{S}\cup\{a\}\right\rangle
\end{array}
$$

Figure 3.7: Cancellation reductions of configuration

Lastly, we examine the exception reductions in Figure 3.8. The [TRY-CATCH] rule spawns two processes, very similar to [SPAWN]. Additionally, it creates another channel *a* of type `exn` to serve as the exception handling context for *P*. The exception handling process *Q* uses channel *a* to wait for exception activation and channel *c* to communicate with process *P*. When a process raises an exception, we activate the exception handling by creating a new process object that sends a label `act` through channel *a*. Similarly, when a process finishes without any exceptions and there is an exception handling channel, we send a label `sil` through channel *a* to terminate the exception handling process.

$$
\begin{array}{rlcl}
[\textsc{try-catch}] & \left\langle\ \mathbf{proc}(\Gamma/\Delta/\Omega/y \twoheadleftarrow \mathbf{try}\ P(y)\ \mathbf{catch}\ Q(y))\ \ ;\mathcal{S}\right\rangle & \longrightarrow & \left\langle\begin{array}{l}\mathbf{proc}(\Gamma_1/\Delta_1,c:A/a:\mathrm{exn}/P(c))\\ \mathbf{proc}(\Gamma_2,a:\mathrm{exn},c:A/\Delta_2/\Omega/\mathrm{erecv}\ a\end{array}\right. \\
[\textsc{activate}] & \left\langle\ \mathbf{proc}(\Gamma/\Delta/a:\mathrm{exn}/\mathbf{raise}\ P)\ \ ;\mathcal{S}\right\rangle & \longrightarrow & \left\langle\begin{array}{l}\mathbf{proc}(\Gamma/\Delta/\cdot/P)\\ \mathbf{proc}(\cdot/a:\mathrm{exn}/\cdot/\mathbf{send}\ a\ \mathrm{act};\mathbf{send}\ a\ (\end{array}\right. \\
[\textsc{silent}] & \left\langle\ \mathbf{proc}(\cdot/\cdot/a:\mathrm{exn}/\cdot)\ \ ;\mathcal{S}\right\rangle & \longrightarrow & \left\langle\ \mathbf{proc}(\cdot/a:\mathrm{exn}/\cdot/\mathbf{send}\ a\ \mathrm{sil};\mathbf{send}\ a\ ()\right.
\end{array}
$$

Figure 3.8: Exception reductions of configuration

3.4 Safety theorems

To ensure the safety of the system, we first need to show the typing of the configuration is preserved as the computation proceeds. Such property is stated in Theorem 3.4.1.

Theorem 3.4.1 (Session Fidelity). *Given a state* $\langle C,\mathcal{S}\rangle$, *if* $\Phi \vdash C :: \Phi'$ *for some* Φ,Φ' *and* $\langle C,\mathcal{S}\rangle \longrightarrow \langle C',\mathcal{S}'\rangle$, *then* $\Phi \vdash C' :: \Phi'$.

The proof of the theorem is casing over the forms of reduction. We will show a few interesting cases to demonstrate how proof proceeds.

Proof. We induct on the forms of reduction.

[SPAWN] In this case, $C = (C_L, \mathbf{proc}(\Gamma/\Delta/\Omega/x \leftarrow P(x);\ Q(x)), C_R)$ for some C_L and C_R. Moreover, we assume the following typing judgements hold.

$$\begin{aligned} \Phi &\vdash C_L :: \Phi_L \\ \Phi_L &\vdash \mathbf{proc}(\Gamma/\Delta/\Omega/x \leftarrow P(x);\ Q(x)) :: \Phi_R \\ \Phi_R &\vdash C_R :: \Phi' \end{aligned}$$

We can split the context Γ and Δ into Γ_1, Γ_2 and Δ_1, Δ_2 based on the meta variables in P and Q. By the reduction rule, we know

$$\begin{aligned} C' &= C_L, \mathbf{proc}(\Gamma_1/a : A, \Delta/\cdot/P(a)), \mathbf{proc}(\Gamma_2, a : A/\Delta_2/\Omega/Q(a)), C_R \\ \mathcal{S}' &= \mathcal{S} \end{aligned}$$

where a is a globally fresh channel. To build the typing judgement for the new state, we apply inversion to the typing

$$\Phi_L \vdash \mathbf{proc}(\Gamma/\Delta/\Omega/x \leftarrow P(x);\ Q(x)) :: \Phi_R$$

We find that for some Φ'_L and A, using the new channel a, we must have

$$\begin{aligned} \Phi_L &= (\Phi'_L, \Gamma_1, \Gamma_2) \\ \Gamma_1 &\vdash P(a) :: \Delta_1, a : A; \cdot \\ \Gamma_2, a : A &\vdash Q(a) :: \Delta_2; \Omega \\ \Phi_R &= (\Phi'_L, \Delta_1, \Delta_2) \end{aligned}$$

Now we can construct a typing derivation for C'

$$\begin{aligned} \Phi &\vdash C_L :: \Phi_L \\ \Phi_L &= (\Phi'_L, \Gamma_1, \Gamma_2) \\ \Phi'_L, \Gamma_1, \Gamma_2 &\vdash \mathbf{proc}(\Gamma_1/a : A, \Delta/\cdot/P(a)) :: (\Phi'_L, \Gamma_2, \Delta_1, a : A) \\ \Phi'_L, \Gamma_2, \Delta_1, a : A &\vdash \mathbf{proc}(\Gamma_2, a : A/\Delta_2/\Omega/Q(a)) :: (\Phi'_L, \Delta_1, \Delta_2) \\ \Phi_R &= (\Phi'_L, \Delta_1, \Delta_2) \\ \Phi_L &\vdash C_R :: \Phi' \end{aligned}$$

This concludes this case of type preservation.

[CANCEL-L] In this case, $C = (C_L, \mathbf{proc}(\Gamma, a : A/\Delta/\Omega/a\lightning; P), C_R)$. We assume the following typing judgements hold.

$$\begin{aligned} \Phi &\vdash C_L :: \Phi_L \\ \Phi_L &\vdash \mathbf{proc}(\Gamma, a : A/\Delta/\Omega/a\lightning; P) :: \Phi_R \\ \Phi_R &\vdash C_R :: \Phi' \end{aligned}$$

By the reduction rule, we know

$$\begin{aligned} C' &= C_L, \mathbf{proc}(\Gamma/\Delta/\Omega/P), C_R \\ \mathcal{S}' &= \mathcal{S} \cup \{a\} \end{aligned}$$

We can apply inversion to the typing judgement

$$\Phi_L \vdash \mathbf{proc}(\Gamma, a : A/\Delta/\Omega/a\lightning; P) :: \Phi_R$$

to find that for some Φ'_L, we must have

$$\begin{aligned} \Phi_L &= (\Phi'_L, \Gamma) \\ \Gamma &\vdash P :: \Delta; \Omega \\ \Phi_R &= (\Phi'_L, \Delta) \end{aligned}$$

Now we can construct a typing derivation for C'

$$\begin{aligned} \Phi &\vdash C_L :: \Phi_L \\ \Phi_L &= (\Phi'_L, \Gamma) \\ \Phi'_L, \Gamma &\vdash \mathbf{proc}(\Gamma/\Delta/\Omega/P) :: (\Phi'_L, \Delta) \\ \Phi_R &= (\Phi'_L, \Delta) \\ \Phi_R &\vdash C_R :: \Phi' \end{aligned}$$

This concludes this case of type preservation.

[SILENT] In this case, $C = (C_L, \mathbf{proc}(\cdot/\cdot/a : \mathrm{exn}/\cdot), C_R)$. We assume the following typing judgements hold.

$$\begin{aligned} \Phi &\vdash C_L :: \Phi_L \\ \Phi_L &\vdash \mathbf{proc}(\cdot/\cdot/a : \mathrm{exn}/\cdot) :: \Phi_R \\ \Phi_R &\vdash C_R :: \Phi' \end{aligned}$$

Applying the inversion, we must have $\Phi_R = \Phi_L$. By the reduction rule, we know

$$\begin{aligned} C' &= C_L, \mathbf{proc}(\cdot/a : \mathrm{exn}/\cdot/\mathbf{send}\ a\ \mathrm{sil}; \mathbf{send}\ a\ ()), C_R \\ \mathcal{S}' &= \mathcal{S} \end{aligned}$$

Since channel a has type $\mathtt{exn}$, we can safely omit the explicit usage of $a : \mathrm{exn}$ and construct a typing derivation for C' as

$$\begin{aligned} \Phi &\vdash C_L :: \Phi_L \\ \Phi_L &\vdash \mathbf{proc}(\cdot/a : \mathrm{exn}/\cdot/\mathbf{send}\ a\ \mathrm{sil}; \mathbf{send}\ a\ ()) :: \Phi_L \\ \Phi_R &= \Phi_L \\ \Phi_R &\vdash C_R :: \Phi' \end{aligned}$$

This concludes this case of type preservation. □

Showing every reduction terminates is not possible in our system because of recursion. For instance, a process `foo` can call itself such that the execution will never terminate. However, we are able to show that well-typed configurations will not get stuck. In other words, a configuration is either final, which means all process objects attempt to communicate along an external channel, or it can reduce to another configuration. We formalize this property in Theorem 3.4.2.

Theorem 3.4.2 (Deadlock Freedom)**.** *Given a state* $\langle C; \mathcal{S}\rangle$ *such that* $\cdot \vdash C :: \Phi$*, then either* C *is final, or* $\langle C; \mathcal{S}\rangle \longrightarrow \langle C'; \mathcal{S}'\rangle$ *for some* $C', \mathcal{S}'$*.*

We require the configuration does not take any input channels. Otherwise, the configuration may get stuck because the messages from the input channel may not be available. The collection of external channels is represented by Φ. Such channels have only one endpoint. In other words, if C satisfies $\cdot \vdash C :: \Phi$, then C is a closed configuration because it does not depend on any channels.

Proof. We prove this theorem by induction on the structure of the configuration, right to left. Since the typing of the configuration requires some ordering, a right to left induction provides useful induction hypothesis. If $C = \cdot$, then C is trivially final and the theorem holds. Now we assume the followings:

$$\begin{aligned} C &= (C_L, \mathbf{proc}(\Gamma/\Delta/\Omega/P)) \\ \cdot &\vdash C_L :: (\Phi', \Gamma) \\ \Phi', \Gamma &\vdash \mathbf{proc}(\Gamma/\Delta/\Omega/P) :: (\Phi', \Delta) \\ \Phi &= (\Phi, \Delta) \\ \cdot &\vdash C :: \Phi \end{aligned}$$

for a process object $\mathbf{proc}(\Gamma/\Delta/\Omega/P)$ and Φ, Φ'.

Now we distinguish the cases by the structure of process P.

[CUT$_A$] In this case $P = x \leftarrow P_1(x);\ P_2(x)$. By induction hypothesis, we know that C_L is either final or can reduce to another configuration C_L' with some cancelled set $\mathcal{S}'$. If it reduces to $\langle C_L'; \mathcal{S}'\rangle$, then $\langle C; \mathcal{S}\rangle \longrightarrow \langle (C_L', \mathbf{proc}(\Gamma/\Delta/\Omega/P)); \mathcal{S}'\rangle$. Otherwise, by reduction, we know $\langle C; \mathcal{S}\rangle \longrightarrow \langle (C_L, \mathbf{proc}(\Gamma_1/a : A, \Delta_1/\cdot/P_1(a)), \mathbf{proc}(\Gamma_2, a : A/\Delta_2/\Omega/P_2(a))); \mathcal{S}\rangle$ with a proper split of the context.

[WEAKENL] In this case $P = a\lightning; P'$ and $\Gamma = \Gamma', a : A$. For simplicity, we assume C_L is final. By reduction, we know $\langle C; \mathcal{S}\rangle \longrightarrow \langle (C_L, \mathbf{proc}(\Gamma'/\Delta/\Omega/P')); \mathcal{S} \cup \{a\}\rangle$.

[EMPTY]-exn In this case $P = \cdot$ while $\Omega = a : \mathsf{exn}$. For simplicity, we assume C_L is final. By reduction, we know $\langle C; \mathcal{S}\rangle \longrightarrow \langle (C_L, \mathbf{proc}(\cdot/a : \mathsf{exn}/\cdot/\mathbf{send}\ a\ \mathsf{sil}; \mathbf{send}\ a\ ())); \mathcal{S}\rangle$.

□

Chapter 4

Interpreter

Experimenting with the proposed type system, we implement a simple interpreter in OCaml. We name the source language EPass, an abbreviation for Exceptional Message Passing. We don't have a target language; instead, we simulate the execution in OCaml and make observations that output to the console screen. There are four main phases in the interpreter: parsing, elaboration, type checking, and runtime simulation. A complete implementation of the interpreter can be found here.

4.1 Grammar

We present the grammar of the EPass language in Appendix A.1. There are a few remarks about the grammar.

- We use `type t =` $\ldots$ for type definition and `proc p` $(\ldots)$ $[\ldots]$ = $\ldots$ for process definition. `exec` p executes a closed process, a process without any input channels, and prints out the observation of the external channels.

- We keep $\multimap$ as the linear implication in our language. Although we do not include it in our system, we have shown $A \multimap B$ can be derived in our system. Thus, $A \multimap B$ is viewed as a syntax sugar that is elaborated to $A^{\perp} \parr B$ during elaboration. We do not include $\perp$, since we can derive it by putting $\mathbf{1}$ in the other side of the context.

- There is not a suitable Unicode character to represent $\parr$. We use @ to represent the multiplicative disjunction.

- We modify the try-catch process such that creating a new channel is not necessary. This can be used when the main process and the exception handling process are independent with each other. Therefore, $y \twoheadleftarrow$ **try** $P(y)$ **catch** $Q(y)$ and **try** P **catch** Q are both valid try-catch processes.

- We distinguish identity and label by putting a ’ character in front of the label. So the type `nat` in EPass looks like

$$\mathtt{nat} \triangleq \oplus\{\text{'zero} : 1, \text{'succ} : \mathtt{nat}\}$$

As checked by Menhir, a LR(1) parser generator for OCaml, there are no ambiguity in the grammar. So every source code that follows the grammar can be parsed into the external language properly. The syntax of the external language can be found in Appendix A.2.

4.2 Elaboration

During the elaboration, we transform the external language into internal language, whose syntax is in Appendix A.3. There are two major goals in this phase.

- We first remove the recursive definitions for types in the external language, which means types have at most one layer. This helps to simplify comparing the equality of two types. When we compare two types, we need to compare the outermost layer, and, if necessary, we can always expand the type by the newly created identity.

  For example, type definitions of `nat` and `list` in the external language

```
type nat = +{'zero : 1, 'succ : nat}
type list = +{'nil : 1, 'cons : nat * list}
```

  will be elaborated to the internal language as

```
type tp_0 = 1
type nat = +{'zero : tp_0, 'succ : nat}
type tp_1 = nat * list
type list = +{'nil : tp_0, 'cons : tp_1}
```

  Besides type elaboration, we also remove the usage of linear application and make distinction between allocated channels and channel variables. Before executing any processes, all the channels should be variables.

- After elaboration, we ensure that every definition is well-formed by checking $\Sigma_{\text{decl}} \vdash \Sigma_{\text{defn}}$. We extract the declarations by scanning the whole program and build the definitions during the elaboration. Type checking errors are reported for any ill-defined processes or types. Additionally, some syntax that may confuse the runtime semantics is also rejected. For instance, executing open processes and having duplicate labels in either type definition or branching continuation are ruled out.

4.3 Runtime simulation

The type checking phase for the interpreter follows the static semantics described in Figure 2.4. We don't have type inference for channels since the grammar enforces type annotations for every channel variable. The runtime simulation essentially follows the reduction rules we discussed before. However, we'd like to make a few remarks on how observations of external channels are made during the reduction.

First, we will demonstrate the reason why we need to make observations. Consider the following example of two processes P_1 and P_2 where a is an external channel and P is some process.

$$\begin{aligned} P_1 &\triangleq \textbf{send}\ a\ k; \textbf{send}\ b\ () \\ P_2 &\triangleq \textbf{recv}\ b\ (() \Rightarrow P) \end{aligned}$$

According to the definition of external channels, channel a has only one endpoint which occurs in process P_1. Therefore, this configuration cannot be further reduced because the interaction between P_1 and P_2 via channel b is blocked by channel a. Considering this as a final configuration in runtime isn't appropriate because process P may have other effects that are not observed. Thus, we need to make observations when the reduction of the configuration gets stuck. In other words, the final configuration in the implementation must be an empty configuration.

We illustrate the output of the program with the following example process that returns the addition of two natural numbers. The type definition can refer to earlier code.

```
proc add (n : nat) [n1 : nat, n2 : nat] =
  recv n1 (
      'zero => recv n1 (() => fwd n n2)
    | 'succ => send n 'succ; call add (n) [n1, n2]
  )

proc add_test_1 (n : nat) [] =
  n1 : nat <- (send n1 'succ; send n1 'zero; send n1 ());
  n2 : nat <- (send n2 'succ; send n2 'succ; send n2 'zero; send n2 ());
  call add (n) [n1, n2]

exec add_test_1
```

The output has the following form:

```
Typecheck successful
Executing process add_test_1:
#0 -> 'succ.'succ.'succ.'zero.()
```

Executing process add_test_1 will generate a fresh variable #0 for the external channel n. The rest represents the message sequence that is sent through the channel. A message sequence is a sequence of observed messages separated by dots, which can be a channel #i, a label 'l, a unit (), or a waiting –. If a channel is in one of the message sequences, then it must have its own sequence. A channel sequence ends with either a unit, a waiting, or a cancellation.

A waiting label is used for an external channel that has a **recv** process which cannot proceed. In the above example, the message sequence 'succ.'succ.'succ.'zero.() for channel #0 represents the natural number 3. In the remainder of this chapter, we'll go through a few examples to see how cancellation and exceptions work.

4.4 Cancellation example

We first demonstrate how to drop resources in the example of returning a length of a list.

```
proc length (len : nat) [l : list] =
  recv l (
      'nil => recv l (() => send len 'zero; send len ())
    | 'cons => recv l (x =>
      cancel x;
      send len 'succ;
      call length (len) [l]
    )
  )
```

The 'nil case is straightforward, where we send 0 through the channel len. In the 'cons case, l has type nat * list. While calculating the length, we do not care the element stored in the list. Therefore, we cancel the channel x and recursively call the process.

Since x is not an external channel in the above example, we cannot see how it is cancelled by observation. Below is an example that simulates a choice of output. We first demonstrate how to drop resources in the example of returning a length of a list.

```
type bool = +{'true : 1, 'false : 1}

proc choice (left : 1, right : 1) [x : bool] =
  recv x (
    'true => recv x (() => cancel right; send left ())
  | 'false => recv x (() => cancel left; send right ())
  )

proc choice_true (left : 1, right : 1) [] =
  tt : bool <- (send tt 'true; send tt ());
  call choice (left, right) [tt]

proc choice_false (left : 1, right : 1) [] =
  ff : bool <- (send ff 'false; send ff ());
  call choice (left, right) [ff]

exec choice_true

exec choice_false
```

Since we know the output forms disjunction, we know exactly only one of the `left` and `right` will provide an output. Which channel provides the output is decided by whether channel *x* is `'true` or `'false`. Processes `choice_true` and `choice_false` tests the `choice` process. We present the output of the program below.

```
Typecheck successful
Executing process choice_true:
#1 -> cancelled
#0 -> ()

Executing process choice_false:
#1 -> ()
#0 -> cancelled
```

As we can see, in `choice_true` process, the `left` channel (#0) receives a unit and the `right` channel (#1) is cancelled. The output is reversed in `choice_false` process, which matches with the expected behavior.

4.5 Exception example

The next example demonstrates how exceptions can be raised and handled. A subtraction of `n1` and `n2` in natural number is not defined if `n2` is bigger than `n1`. We deal such case by raising an exception and dropping all the unused resources.

```
proc subtract (n : nat) [n1 : nat, n2 : nat] =
  recv n1 (
    'zero => recv n1 (() =>
      recv n2 (
        'zero => recv n2 (() => send n 'zero; send n ())
      | 'succ => raise (cancel n; cancel n2)
      )
    )
  | 'succ => recv n2 (
      'zero => recv n2 (() => send n 'succ; fwd n n1)
    | 'succ => call subtract (n) [n1, n2]
    )
  )

proc wrap_subtract (n : nat, n_exn : nat) [n1 : nat, n2 : nat] =
  try call subtract (n) [n1, n2]
  catch send n_exn 'zero; send n_exn ()
```

We provide a wrapper process `wrap_subtract` to invoke the `subtract` process, as we don't allow exceptions without any handling process. Since the process is for example usage, we don't perform any significant action in the `catch` branch. Instead, we simply return 0 from another external channel `n_exn`.

Information can also be exchanged between the main process and the exception handling process by creating a new channel. For instance, when zipping two lists and the lengths of the lists are not equal, the first attempt to zip will raise an exception and pass the difference in lengths of the lists to the second attempt, which handles the exception. The second attempt can then adjust for the length difference and reattempt zipping the remaining parts of the lists.

Chapter 5

Other features

In this chapter, we delve into advanced features and remove some constraints in the system to enhance the language's capabilities in various situations. We study uncaught exceptions and non-exhaustive matching in our system. To accommodate these changes, we revisit the typing judgement and the reduction rules. Moreover, we restate the safety theorem to align with the new system. Related work and future plans are also included in the last part of this chapter.

5.1 Uncaught exception

In most programming languages, uncaught exceptions cause runtime crashes and are reported from the top level. For example, the following expression

```
let x = 1 / 0 in
x + 1
```

aborts the program and print the following information into console.

```
Uncaught exception:

Division_by_zero

Raised at ... in file "...", line ..., characters ...
Called from ... in file "...", line ..., characters ...
```

Such behavior is prohibited in our system, since we require every exception to be caught by a process. Otherwise, the program fails at the typing checking phase. However, we can relax this constraint by allowing uncaught exceptions. We modify the [RAISE] rule.

$$[\textsc{raise'}]\quad \frac{\Gamma \vdash P :: \Delta; \cdot}{\Gamma \vdash \mathbf{raise}\ P :: \Delta; \Omega}$$

We relax the constraint that Ω must be a singleton set. In [RAISE'] rule, Ω can be either an empty set or a singleton set. The premise does not change since **raise** P uses the exception channel no matter whether it is in Ω or not. A key observation is that under such relaxation, judgement $\Gamma \vdash P :: \Delta; x : \mathrm{exn}$ is provable if and only if $\Gamma \vdash P :: \Delta; \cdot$. Thus, it seems that the Ω context is useless and can be omitted in the typing judgement. However, we can utilize Ω during the type checking phase and output warning information if an exception does not have direct handling process. For instance, the following code

```
proc foo (x : 1) [] =
  raise (cancel x)
```

will output the following message to the console.

```
  Warning: In process foo, raise (cancel x) does not have
  its corresponding exceptional handler
```

However, the warning message is conservative in some sense. For example, if another process bar calls the process foo. No matter whether bar catches the exception or not, the warning message still prints to the console. Only direct caught exceptions like

```
proc foo' (x : 1, y : 1) [] =
  try raise (cancel x) catch send y ()
```

will not produce the warning message.

We adapt the same industry standard for uncaught exceptions. If an exception is not caught by any process, the program will abort. We define an error configuration Λ in Figure 5.1 to indicate that the program is terminated unexpectedly.

$$\begin{array}{llll} \text{Configuration} & C & ::= & \ldots \\ & & \mid & \Lambda \qquad \text{error configuration} \end{array}$$

$$[\textsc{error}]\quad \frac{}{\Phi \vdash \Lambda :: \Phi'}$$

$$[\textsc{uncaught}]\ \langle\ \mathbf{proc}(\Gamma/\Delta/\cdot/\mathbf{raise}\ P)\ ; \mathcal{S}\rangle \longrightarrow \langle\ \Lambda\ ; \mathcal{S}\rangle$$

Figure 5.1: Extended syntax, typing judgement, and reduction of the configuration

The typing judgment for the error configuration does not matter. Therefore, Λ is well-typed for any input context Φ and output context Φ'. We add a new rule [UNCAUGHT], which is another case of the [ACTIVATE] rule where Ω is empty. When an exception is raised with no handling channels, the configuration steps to the error configuration Λ directly. This avoids the problem of concurrent top-level exceptions where it is not clear which exception should be raised. In our system, the configuration reduces to the only error configuration, and other processes are stopped.

To account for the new reduction rule, we revisit the safety theorems and their proofs. There is no major change in the proofs of Theorem 3.4.1, because if a configuration reduces to an error configuration, we know such configuration is still well-typed by [ERROR]. We revise Theorem 3.4.2 as follows.

Theorem 5.1.1 (Deadlock Freedom Revisited)**.** *Given a state* $\langle C; \mathcal{S} \rangle$ *such that* $\cdot \vdash C :: \Phi$*, then either* C *is final,* $\langle C; \mathcal{S} \rangle \longrightarrow \langle \Lambda; \mathcal{S} \rangle$ *or* $\langle C; \mathcal{S} \rangle \longrightarrow \langle C'; \mathcal{S}' \rangle$ *for some* $C', \mathcal{S}'$*.*

The proof of this theorem extends the original proof by adding the case of uncaught exception. The proof is trivial and thus omitted.

In the interpreter, we abort the program simulation when we face with an uncaught exception. For example, in the following tiny example,

```
proc foo (x : 1) [] =
  raise (cancel x)

proc foo1 (x : 1) [] =
  send x ()

exec foo1

exec foo2
```

we get the following message from the console.

```
  Executing process foo1:
  Uncaught exception, process aborted

  Executing process foo2:
  #0 -> ()
```

It is worth noting that execution of processes are independent of each other, thus a crash in `foo1` does not affect the execution of `foo2`. The next feature builds on the idea of uncaught exceptions.

5.2 Non-exhaustive match

Pattern matching is commonly used for matching on algebraic data types in functional programming languages. The label message reduction in our system resembles pattern matching. For instance, when receiving messages from a channel of type `nat`, we need to provide a branch for each label in the type definition. Currently, we require that every label must have its own branch as restricted by the [$\oplus L$] and [$\& R$] rules. Non-exhaustive matching is a common warning or error in most programming languages. Such matching will halt program execution if the expression does not match any of the branches. A good programming practice is to use a wildcard case to handle all remaining, possibly uninteresting cases. We implement the idea of non-exhaustive matching in our system by raising an exception when a label does not have its corresponding branch. To relax such constraints, we modify the [$\oplus L$] and [$\& R$] rules in Figure 5.2. The branching label collection L' is a subset of the original label collection L, which means in the original rule, $L' = L$.

$$[\oplus L'] \quad \frac{\Gamma, x : A_l \vdash P_l :: \Delta; \Omega \qquad L' \subseteq L \qquad (\forall l \in L')}{\Gamma, x : \oplus\{l : A_l\}_{l \in L} \vdash \mathbf{recv}\ x\ (l \Rightarrow P_l)_{l \in L'} :: \Delta; \Omega}$$

$$[\& R'] \quad \frac{\Gamma \vdash P_l :: x : A_l, \Delta; \Omega \qquad L' \subseteq L \qquad (\forall l \in L')}{\Gamma \vdash \mathbf{recv}\ x\ (l \Rightarrow P_l)_{l \in L'} :: x : \&\{l : A_l\}_{l \in L}, \Delta; \Omega}$$

Figure 5.2: Refined typing judgement for non-exhaustive match

The original reduction rules for labels no longer hold. In [LABEL-L] and [LABEL-R] rules, we need to ensure that $k \in L$. To handle the possibility of $k \notin L$, we add two new reduction rules in Figure 5.3. The intended behavior is to raise an exception and channel all channels prepared for the **recv** process.

$$[\text{LABEL-R'}] \left\langle \begin{array}{l} \mathbf{proc}(\Gamma_1 / a : \oplus\{l : A_l\}_{l \in L}, \Delta_1 / \Omega_1 / \mathbf{send}\ a\ k; P) \\ \mathbf{proc}(\Gamma_2, a : \oplus\{l : A_l\}_{l \in L} / \Delta_2 / \Omega_2 / \mathbf{recv}\ a\ (l \Rightarrow P_l)_{l \in L}) \end{array} (k \notin L); \mathcal{S} \right\rangle \longrightarrow \left\langle \begin{array}{l} \mathbf{proc}(\Gamma_1 / a : A_k, \Delta_1 / \Omega \\ \mathbf{proc}(\Gamma_2, a : A_k / \Delta_2 / \Omega \end{array} \right.$$

$$[\text{LABEL-L'}] \left\langle \begin{array}{l} \mathbf{proc}(\Gamma_1, a : \&\{l : A_l\}_{l \in L} / \Delta_1 / \Omega_1 / \mathbf{send}\ a\ k; P) \\ \mathbf{proc}(\Gamma_2 / a : \&\{l : A_l\}_{l \in L}, \Delta_2 / \Omega_2 / \mathbf{recv}\ a\ (l \Rightarrow P_l)_{l \in L}) \end{array} (k \notin L); \mathcal{S} \right\rangle \longrightarrow \left\langle \begin{array}{l} \mathbf{proc}(\Gamma_1, a : A_k / \Delta_1 / \Omega \\ \mathbf{proc}(\Gamma_2 / a : A_k, \Delta_2 / \Omega \end{array} \right.$$

Figure 5.3: Extended standard reduction rules of configuration for non-exhaustive match

In the runtime simulation, a warning message will be printed to the console any branching continuation misses a label. A non-exhaustive match, if not handled, will raise an uncaught exception and behaves as described in the previous section. We demonstrate the non-exhaustive match by the following simple example.

```
type choice = +{'a : 1, 'b : 1, 'c : 1}

proc choose (x : bool) [c : choice] =
    recv c (
        'a => send x 'true; fwd x c
      | 'c => send x 'false; fwd x c
    )

proc test_choose1 (x : bool) [] =
    c : choice <- (send c 'a; send c ());
    call choose (x) [c]

proc test_choose2 (x : bool) [] =
    c : choice <- (send c 'b; send c ());
    call choose (x) [c]

exec test_choose1

exec test_choose2
```

The output of the program is as follows.

```
Warning: In process choose, choice type of channel c
has non-exaustive match, missing label: 'b

Executing process test_choose1:
#0 -> 'true.()

Executing process test_choose2:
Uncaught exception, process aborted
```

A warning message is printed for the process choose because the label `’b` is missing. In the process `test_choose2`, an uncaught exception is raised because the label send through channel `c` is `’b` and the program is aborted. Such feature can also be used to simulate process assertion. If we are certain that some label should never be sent through a channel, for instance `’true` or `’false`, we can omit such cases in branching.

5.3 Related work and future work

We divide the related work into two parts: channel cancellation and exception handling.

Channel cancellation was first proposed in Affine session that models channel interruption. We utilize this idea and refine it as the proof term of explicit weakening rules. The refinement provides a better way to understand channel cancellation and connects the relation between processed and sequent proof terms.

In the work of Conversation Calculus, the model provides a simple mechanism by supporting two primitives, `throw`, and `try catch`, to handle exceptional situations [VCS08]. Compared with our work, we provide a typed language where exceptions are generalized. For instance, we can simulate server timeout and client reconnect examples studied by Brun and Dardha [BD23].

Adjoint logic generalizes the usual session types by supporting multiple modes of communication [PP19]. The current system works under affine logic, which is one layer of adjoint logic. It remains interesting how exception handling can be generalized to adjoint logic. In other words, how to pass exceptions from one mode to another mode while both have the correct intended behavior and maintain the safety of the language. Meanwhile, previous works model data layout using semi-axiomatic sequent calculus [DP23]. Memory access errors such as writing to a freed memory cell or dereferencing an invalid address. In future work, we plan to work under a shared memory interpretation and study the exceptions.

Appendix A

EPass

A.1 Grammar

```
<idchar> ::= [a-zA-Z_0-9']*
<id> ::= [a-zA-Z_]<idchar>
<tag> ::= '<idchar>+

<tp> ::= <id>
       | '1'
       | '+' '{' <alts> '}'
       | '&' '{' <alts> '}'
       | <tp> '@' <tp>
       | <tp> '*' <tp>
       | <tp> '-o' <tp>
       | '(' <tp> ')'

<altsfollow> ::= · | ',' <tag> ':' <tp> <altsfollow>

<alts> ::= <tag> ':' <tp> <altsfollow>

<procfollow> ::= · | ';' <proc>

<simpleproc> ::= 'send' <id> <msg>
         | 'recv' <id> '(' <cont> ')'
         | 'fwd' <id> <id>
         | 'call' <id> '(' <args> ')' '[' <args> ']'
         | 'cancel' <id>
```

```
<spawnopt> ::= · | <id> ':' <tp> '<<-'

<proc> ::= 'send' <id> <msg> <procfollow>
         | 'recv' <id> '(' <cont> ')'
         | 'fwd' <id> <id>
         | 'call' <id> '(' <args> ')' '[' <args> ']'
         | 'cancel' <id> <procfollow>
         | <spawnopt> 'try' <proc> 'catch' <proc>
         | 'raise' <proc>
         | <id> ':' <tp> '<-' <simpleproc> ';' <proc>
         | <id> ':' <tp> '<-' '(' <proc> ')' ';' <proc>
         | '(' <proc> ')'

<argsfollow> ::= · | ',' <id> <argsfollow>

<args> ::= <id> <argsfollow>

<annoargsfollow> ::= · | ',' <id> ':' <type> <annoargsfollow>

<annoargs> ::= <id> ':' <type> <annoargsfollow>

<contfollow> ::= · | '|' <msg> '=>' <proc> <contfollow>

<cont> ::= <msg> '=>' <proc> <contfollow>

<msg> ::= '(' ')'
        | <tag>
        | <id>

<defn> ::= 'type' <id> = <tp>
         | 'proc' <id> '(' <annoargs> ')' '[' <annoargs> ']' '=' <proc>
         | 'exec' <id>

<prog> ::= · | <defn> <prog>
```

A.2 External language

```
type typ =
  | Var of string
  | Tensor of typ * typ
  | Par of typ * typ
  | Plus of (string * typ) list
  | With of (string * typ) list
  | One
  | Lolli of typ * typ

type msg =
  | Unit
  | Label of string
  | Channel of string

type proc =
  | Send of string * msg * proc option
  | Recv of string * cont
  | Fwd of string * string
  | Call of string * string list * string list
  | Cancel of string * proc option
  | Trycatch of (string * typ) option * proc * proc
  | Raise of proc
  | Cut of string * typ * proc * proc

and cont =
  | ContUnit of proc
  | ContLabel of (string * proc) list
  | ContChannel of string * proc

type def =
  | TypDef of string * typ
  | ProcDef of string * (string * typ) list * (string * typ) list * proc
  | Exec of string

type prog = def list
```

A.3 Internal language

```
type channel =
  | ChanVar of string
  | ChanConst of int

type typ =
  | Tensor of string * string
  | Par of string * string
  | Plus of (string * string) list
  | With of (string * string) list
  | One
  | Dual of string

type msg =
  | Unit
  | Label of string
  | Channel of channel

type proc =
  | Send of channel * msg * proc option
  | Recv of channel * cont
  | Fwd of channel * channel
  | Call of string * channel list * channel list
  | Cancel of channel * proc option
  | Trycatch of (channel * typ) option * proc * proc
  | Raise of proc
  | Cut of channel * typ * proc * proc
  | Null

and cont =
  | ContUnit of proc
  | ContLabel of (string * proc) list
  | ContChannel of channel * proc

type def =
  | TypDef of string * typ
  | ProcDef of string * (channel * typ) list * (channel * typ) list * proc
  | Exec of string
```

```
type prog = def list
```

Bibliography

[Abr93] Samson Abramsky. “Computational interpretations of linear logic”. In: *Theoretical Computer Science* 111.1 (1993), pp. 3–57. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(93)90181-R. URL: https://www.sciencedirect.com/science/article/pii/030439759390181R.

[BD23] Matthew Alan Le Brun and Ornela Dardha. *MAGπ: Types for Failure-Prone Communication*. 2023. arXiv: 2301.10827 [cs.PL].

[CPT14] Luís Caires, Frank Pfenning, and Bernardo Toninho. “Linear Logic Propositions as Session Types”. In: *Mathematical Structures in Computer Science* 760 (Nov. 2014). DOI: 10.1017/S0960129514000218.

[DP23] Henry DeYoung and Frank Pfenning. “Data Layout from a Type-Theoretic Perspective”. In: *Electronic Notes in Theoretical Informatics and Computer Science* Volume 1-Proceedings of... (Feb. 2023). ISSN: 2969-2431. DOI: 10.46298/entics.10507. URL: http://dx.doi.org/10.46298/entics.10507.

[Fri78] Harvey Friedman. “Classically and intuitionistically provably recursive functions”. In: *Higher Set Theory*. Ed. by Gert H. Müller and Dana S. Scott. Berlin, Heidelberg: Springer Berlin Heidelberg, 1978, pp. 21–27. ISBN: 978-3-540-35749-0.

[How80] William Alvin Howard. “The Formulae-as-Types Notion of Construction”. In: *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Ed. by Haskell Curry, Hindley B., Seldin J. Roger, and P. Jonathan. Academic Press, 1980.

[Mil+97] Robin Milner, Robert Harper, David MacQueen, and Mads Tofte. *The Definition of Standard ML*. The MIT Press, May 1997. ISBN: 9780262287005. DOI: 10.7551/mitpress/2319.001.0001. URL: https://doi.org/10.7551/mitpress/2319.001.0001.

[MV14] Dimitris Mostrous and Vasco Thudichum Vasconcelos. "Affine Sessions". In: *Coordination Models and Languages*. Ed. by Eva Kühn and Rosario Pugliese. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 115–130. ISBN: 978-3-662-43376-8.

[PP19] Klaas Pruiksma and Frank Pfenning. "A Message-Passing Interpretation of Adjoint Logic". In: *Proceedings Programming Language Approaches to Concurrency-and Communication-cEntric Software, PLACES@ETAPS 2019, Prague, Czech Republic, 7th April 2019*. Ed. by Francisco Martins and Dominic Orchard. Vol. 291. EPTCS. 2019, pp. 60–79. DOI: 10.4204/EPTCS.291.6. URL: https://doi.org/10.4204/EPTCS.291.6.

[VCS08] Hugo Vieira, Luís Caires, and João Seco. "The Conversation Calculus: A Model of Service-Oriented Computation". In: Mar. 2008, pp. 269–283. ISBN: 978-3-540-78738-9. DOI: 10.1007/978-3-540-78739-6_21.

[Wad12] Philip Wadler. "Propositions as sessions". In: *SIGPLAN Not.* 47.9 (Sept. 2012), pp. 273–286. ISSN: 0362-1340. DOI: 10.1145/2398856.2364568. URL: https://doi.org/10.1145/2398856.2364568.