

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

Acknowledgements	iv
Contents	v
1 Introduction	1
1.1 Proofs as processes	2
1.2 Classical linear logic	3
1.3 Affine logic	4
2 Type system	6
2.1 Propositions	6
2.2 Processes	9
2.3 Exception processes	12
3 Runtime semantics	14
3.1 Configuration	14
3.2 Recursion	16
3.3 Execution	17
A Proofs	18
Bibliography	19

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.

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

where variables x_1, x_2, \dots, x_n, c represents distinct channels, propositions A_1, A_2, \dots, 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.

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

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 $A \supset \perp$.

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

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.

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

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.

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

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

2.1 Propositions

The syntax of the propositions are defined in [Figure 2.1](#).

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

Figure 2.1: Syntax of propositions

$(A)^\perp$	\longrightarrow	A^\perp	A atom
$\mathbf{1}^\perp$	\longrightarrow	\perp	
\perp^\perp	\longrightarrow	$\mathbf{1}$	
$(A \otimes B)^\perp$	\longrightarrow	$A^\perp \wp B^\perp$	
$(A \wp B)^\perp$	\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	$\oplus\{l : A_l^\perp\}_{l \in L}$	

Figure 2.2: Dual operations on propositions

The propositions can be classified into two categories: multiplicatives (\otimes, \wp) and additives ($\oplus, \&$). The behavior of each type based 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 antecedent is as 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 simply the explanation of the propositions, we discuss its logical behavior of each proposition in the succedent through its inference right rule.

- 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

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

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

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

The multiplicative disjunction $A \wp B$ 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 $A^\perp \wp B$. 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 $\wp R$, we conclude $\Gamma \vdash A \wp B$. Since the proof is invertible, we can derive the other direction as well.

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

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

The identity rule closes the proof by ensuring that there is only one same 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.

$$\frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta} \text{weakenL} \quad \frac{\Gamma \vdash \Delta}{\Gamma \vdash A, \Delta} \text{weakenR}$$

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:

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

- 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:

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

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

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

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](#).

Messages	M	$::=$	$()$
			$ $
			l
			$ $
			x
Continuations	K	$::=$	$((\ () \Rightarrow P)$
			$ $
			$(l \Rightarrow P_l)_{l \in L}$
			$ $
			$(x \Rightarrow P(x))$
Processes	P, Q	$::=$	send x M
			$ $
			recv x K
			$ $
			fwd y x
			$ $
			$x \nmid$
			$ $
			$x \leftarrow P(x); Q(x)$
			$ $
			raise M
			$ $
			$x \leftarrow \mathbf{try} P(x) \mathbf{catch} Q(x)$
			$ $
			$P; Q$

Figure 2.3: Syntax of processes

We separate the terms into three categories: messages, continuations, and processes. We use variables to represent channels and strings 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 the information flows through channels. Processes of **send** and **recv** are the 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 send or receive message depends on the polarity of the connective. Positive connectives, whose right rule is not invertible, will send a message in its right rule and receive a message in its left rule. The intuition is that non-invertible rules always make a choice, while invertible rules does not contain any extra information.

The proposition restricts the message type a channel can send or receive. A channel of type $\mathbf{1}$ or \perp can only send or receive a unit message, respectively. In logic system, $\mathbf{1}$ represents the empty truth, and \perp 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 unit message to computation suspension in functional programming. Under call-by-value setting, an expression e needs to be eagerly evaluated before it can be involved in future computation. However, if we replace e by $\lambda(x : \text{unit}.e)$, 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 \wp can send or receive a channel and its type evolved 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 evolved to type B . The last message corresponds to choice type \oplus and $\&$. A choice type is a collection of types distinguished by labels. For instance, type `bool` can be written as $\oplus\{\text{true} : \mathbf{1}, \text{false} : \mathbf{1}\}$, 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 collection L . We add a subscript k as a label to the $[\oplus R_k]$ rule to indicate the process chooses k . The type of the channel evolved from choice type to the corresponding type 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

$$\begin{array}{c}
\begin{array}{c}
\text{[1R]} \\
\hline
\cdot \vdash \mathbf{send} \ x \ () :: x : \mathbf{1}; \Omega
\end{array}
\qquad
\begin{array}{c}
\text{[1L]} \\
\hline
\Gamma \vdash P :: \Delta; \Omega \\
\hline
\Gamma, x : \mathbf{1} \vdash \mathbf{recv} \ x \ (() \Rightarrow P) :: \Delta; \Omega
\end{array}
\\[10pt]
\begin{array}{c}
\text{[\perp R]} \\
\hline
\Gamma \vdash P :: \Delta; \Omega \\
\hline
\Gamma \vdash \mathbf{recv} \ x \ (() \Rightarrow P) :: x : \perp, \Delta; \Omega
\end{array}
\qquad
\begin{array}{c}
\text{[\perp L]} \\
\hline
x : \perp \vdash \mathbf{send} \ x \ () :: \cdot; \Omega
\end{array}
\\[10pt]
\begin{array}{c}
\text{[\otimes R*]} \\
\hline
\Gamma \vdash P :: x : B, \Delta; \Omega \\
\hline
\Gamma, y : A \vdash \mathbf{send} \ x \ y; P :: x : A \otimes B, \Delta; \Omega
\end{array}
\qquad
\begin{array}{c}
\text{[\otimes L]} \\
\hline
\Gamma, x : B, y : A \vdash P :: \Delta; \Omega \\
\hline
\Gamma, x : A \otimes B \vdash \mathbf{recv} \ x \ (y \Rightarrow P(y)) :: \Delta; \Omega
\end{array}
\\[10pt]
\begin{array}{c}
\text{[\wp R]} \\
\hline
\Gamma \vdash P :: \Delta; x : B, y : A, \Omega \\
\hline
\Gamma \vdash \mathbf{recv} \ x \ (y \Rightarrow P(y)) :: x : A \wp B, \Delta; \Omega
\end{array}
\qquad
\begin{array}{c}
\text{[\wp L*]} \\
\hline
\Gamma, x : B \vdash P :: \Delta; \Omega \\
\hline
\Gamma, x : A \wp B \vdash \mathbf{send} \ x \ y; P :: y : A, \Delta; \Omega
\end{array}
\\[10pt]
\begin{array}{c}
\text{[\oplus R}_k\text{]} \\
\hline
\Gamma \vdash P :: A_k; \Omega \quad k \in L \\
\hline
\Gamma \vdash \mathbf{send} \ x \ k; P :: x : \oplus \{ l : A_l \}_{l \in L}, \Delta; \Omega
\end{array}
\qquad
\begin{array}{c}
\text{[\oplus L]} \\
\hline
\Gamma, x : A_l \vdash P_l :: \Delta; \Omega \quad (\forall l \in L) \\
\hline
\Gamma, x : \oplus \{ l : A_l \}_{l \in L} \vdash \mathbf{recv} \ x \ (l \Rightarrow P_l)_{l \in L} :: \Delta; \Omega
\end{array}
\\[10pt]
\begin{array}{c}
\text{[\& R]} \\
\hline
\Gamma, x : A_l \vdash P_l :: \Delta; \Omega \quad (\forall l \in L) \\
\hline
\Gamma, x : \& \{ l : A_l \}_{l \in L} \vdash \mathbf{recv} \ x \ (l \Rightarrow P_l)_{l \in L} :: \Delta; \Omega
\end{array}
\qquad
\begin{array}{c}
\text{[\& L}_k\text{]} \\
\hline
\Gamma, x : A_k \vdash P :: \Delta; \Omega \quad k \in L \\
\hline
\Gamma, x : \& \{ l : A_l \}_{l \in L} \vdash \mathbf{send} \ x \ k; P :: \Delta; \Omega
\end{array}
\\[10pt]
\begin{array}{c}
\text{[WEAKENR]} \\
\hline
\Gamma \vdash P :: \Delta; \Omega \\
\hline
\Gamma \vdash x \not\downarrow; P :: x : A, \Delta; \Omega
\end{array}
\qquad
\begin{array}{c}
\text{[WEAKENL]} \\
\hline
\Gamma \vdash P :: \Delta; \Omega \\
\hline
\Gamma, x : A \vdash x \not\downarrow; P :: \Delta; \Omega
\end{array}
\qquad
\begin{array}{c}
\text{[IDENTITY}_A\text{]} \\
\hline
y : A \vdash \mathbf{fwd} \ x \ y :: x : A; \Omega
\end{array}
\\[10pt]
\begin{array}{c}
\text{[CUT}_A\text{]} \\
\hline
\Gamma_1 \vdash P(x) :: x : A, \Delta_1; \cdot \quad \Gamma_2, x : A \vdash Q(x) :: \Delta_2; \Omega \\
\hline
\Gamma_1, \Gamma_2 \vdash x \leftarrow P(x); Q(x) :: \Delta_1, \Delta_2; \Omega
\end{array}
\\[10pt]
\begin{array}{c}
\text{[RAISE]} \\
\hline
\Gamma \vdash P :: \Delta; \cdot \\
\hline
\Gamma \vdash \mathbf{raise} \ P :: \Delta; \Omega
\end{array}
\\[10pt]
\begin{array}{c}
\text{[TRYCATCH}_A\text{]} \\
\hline
\Gamma_1 \vdash P(y) :: y : A, \Delta_1; x : \text{exn} \quad \Gamma_2, x : \text{exn}, y : A \vdash \mathbf{erecv} \ x \ Q(y) :: \Delta_2; \Omega \\
\hline
\Gamma_1, \Gamma_2 \vdash y \leftarrow \mathbf{try} \ P(y) \ \mathbf{catch} \ Q(y) :: \Delta_1, \Delta_2; \Omega
\end{array}
\end{array}$$

Figure 2.4: Static semantics of the type system

the later section.

We now discuss the structural rules of our system. The weakening rule is represented by the channel cancellation process $x \cancel{\downarrow}$. Although we have two different weakening rules : $[\text{WEAKENL}]$ and $[\text{WEAKENR}]$, we do not distinct its difference in process. The process $x \cancel{\downarrow}$ discards the channel x from the context and continues with process P under the remaining context. The $[\text{IDENTITY}_A]$ rule is represented by the process $\mathbf{fwd} \ y \ x$, 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 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 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, this 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:

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

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 `[TRYCATCHA]`. 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:

$$\begin{aligned} \Gamma \vdash \text{erecv } x \ Q(y) :: \Delta; \Omega \triangleq \Gamma \vdash \mathbf{recv } x \ (\text{act} \Rightarrow \mathbf{recv } x \ (\ () \Rightarrow Q(y)) \\ | \text{sil} \Rightarrow \mathbf{recv } x \ (\ () \Rightarrow y\downarrow; \Gamma\downarrow; \Delta\downarrow)) :: \Delta; \Omega \end{aligned}$$

It waits a label send 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 side, if the main process executes finished without any exceptions, a `sil` label will lead to a branch of cancelling all resources prepared for the exception handling. The exception handling context is inherited from the original process.

Now we go back to 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 value, we raise a process as described in `[RAISE]`. `raise P` raises an exception and continue to execute with process P . Right now, we require the exception to be caught by the exception handling process. This invariant is enforced by requires Ω is non-empty. The restriction can be relaxed, since uncaught exceptions are common in programming languages. We explore this feature in the later chapter.

Chapter 3

Runtime semantics

3.1 Configuration

To formalize the runtime semantics, we can view each single 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 and adding new process object into the configuration. We define the configuration C in [Figure 3.1](#).

Configuration	C	$::=$	\cdot	empty configuration
			$ C_1, C_2$	join configuration
			$ \mathbf{proc}(\Gamma/\Delta/\Omega/P)$	process object

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](#).

$$\begin{array}{c}
\text{[EMPTY]} \quad \text{[JOIN]} \\
\frac{}{\Phi \vdash \cdot :: \Phi} \quad \frac{\Phi \vdash C_1 :: \Phi' \quad \Phi' \vdash C_2 :: \Phi''}{\Phi \vdash C_1, C_2 :: \Phi''} \\
\\
\text{[OBJECT]} \\
\hline
\Phi, \Gamma \vdash \mathbf{proc}(\Gamma/\Delta/\Omega/P) :: \Delta, \Phi
\end{array}$$

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. We do also want such feature in our system. However, the formalization of recursion has little to do with logics. We achieve such result by having a global signature Σ . This signature records both type and process definition. A type definition has the form $t = A$, where t is a type identifier and A is a proposition. Process definition has the form $p(y_1, y_2, \dots, y_m)[x_1, x_2, \dots, x_n] = P$. The identification includes a process name p , a list of output channels y_1, y_2, \dots, y_m and a list of input channels x_1, x_2, \dots, x_n . The process such identification represents is P . We use the judgement $\Sigma \vdash \Sigma' \mathbf{sig}$ to represent that Σ' is a valid signature given all the declarations in Σ . We present the rules for signature in [Figure 3.3](#). The judgement for well-formed type $\Sigma \vdash A \mathbf{type}$ and well-formed process $\Sigma \vdash P \mathbf{proc}$ can be defined on the structure of propositions and processes, respectively, and thus we omit the rules here.

$$\begin{array}{c}
 \begin{array}{c} \text{[EMPTY]} \\ \hline \Sigma \vdash \cdot \mathbf{sig} \end{array}
 \quad
 \begin{array}{c} \text{[TYPEDEF]} \\ \hline \Sigma \vdash \Sigma' \mathbf{sig} \quad \Sigma \vdash A \mathbf{type} \\ \hline \Sigma \vdash \Sigma', t = A \mathbf{sig} \end{array} \\
 \\
 \begin{array}{c} \text{[PROCDEF]} \\ \hline \Sigma \vdash \Sigma' \mathbf{sig} \quad \Sigma \vdash P(x_1, x_2, \dots, x_n, y_1, y_2, \dots, y_m) \mathbf{proc} \\ \hline \Sigma \vdash \Sigma', p(y_1, y_2, \dots, y_m)[x_1, x_2, \dots, x_n] = P \mathbf{sig} \end{array}
 \end{array}$$

Figure 3.3: Judgement of well-formed signature

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

- Natural number `nat`

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

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`

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

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 3.4. A process can call itself to achieve self recursion.

$$\begin{array}{l}
 \text{Processes } P, Q ::= \dots \\
 \quad | \quad \mathbf{call} \, p \, (y_1, y_2, \dots, y_m) \, [x_1, x_2, \dots, x_n] \\
 \\
 \frac{[\text{CALL}] \quad \Sigma \vdash p(y_1, y_2, \dots, y_m)[x_1, x_2, \dots, x_n] = P \, \mathbf{sig} \quad \Gamma \vdash P :: \Delta; \Omega}{\Gamma \vdash \mathbf{call} \, p \, (y_1, y_2, \dots, y_m) \, [x_1, x_2, \dots, x_n] :: \Delta; \Omega} \\
 \\
 \text{where } \Gamma = x_1 : A_1, x_2 : A_2, \dots, x_n : A_n \text{ and } \Delta = y_1 : C_1, y_2 : C_2, \dots, y_m : C_m
 \end{array}$$

Figure 3.4: Extended syntax and static semantics of the type system

The **call** process looks up the signature Σ 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 Execution

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 $\langle C; S \rangle \longrightarrow \langle C'; S' \rangle$. The reduction rules only examine parts of the configuration. The untouched parts are preserved in the new configuration. We first present the standard reduction rule without exceptions in Figure 3.5.

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

Figure 3.5: Selected configuration reduction rules

Appendix A

Proofs

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](https://doi.org/10.1016/0304-3975(93)90181-R). URL: <https://www.sciencedirect.com/science/article/pii/030439759390181R>.
- [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](https://doi.org/10.1017/S0960129514000218).
- [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](https://doi.org/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.
- [Wad12] Philip Wadler. “Propositions as sessions”. In: *SIGPLAN Not.* 47.9 (Sept. 2012), pp. 273–286. ISSN: 0362-1340. DOI: [10.1145/2398856.2364568](https://doi.org/10.1145/2398856.2364568). URL: <https://doi.org/10.1145/2398856.2364568>.