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

A Proofs 11

Bibliography 12

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.

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

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.

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

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.

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

This judgement can be viewed as a process P that communicates along channels $x_1, x_2, \ldots, x_n$ and $y_1, y_2, \ldots, y_m$, 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

2.1 Propositions

The syntax of the propositions are defined in Figure 2.1.

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

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 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

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

  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 reversible, we can derive the other direction as well.

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

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

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

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

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:

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

- 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:

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

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.

$$
\begin{array}{rrcl}
\text{Messages} & M & ::= & () \\
 & & \mid & l \\
 & & \mid & x \\
\text{Continuations} & K & ::= & (() \Rightarrow P) \\
 & & \mid & (l \Rightarrow P_l)_{l \in L} \\
 & & \mid & (x \Rightarrow P(x)) \\
\text{Processes} & P, Q & ::= & \mathbf{send}\ x\ M \\
 & & \mid & \mathbf{recv}\ x\ K \\
 & & \mid & \mathbf{fwd}\ y\ x \\
 & & \mid & x \not\leftarrow \\
 & & \mid & x \leftarrow P(x);\ Q(x) \\
 & & \mid & \mathbf{raise}\ M \\
 & & \mid & x \twoheadleftarrow \mathbf{try}\ P(x)\ \mathbf{catch}\ Q(x) \\
 & & \mid & P; Q
\end{array}
$$

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

$$[\mathbf{1}R] \quad \frac{}{\cdot \vdash \mathbf{send}\ x\ () :: x : \mathbf{1}; \Omega} \qquad [\mathbf{1}L] \quad \frac{\Gamma \vdash P :: \Delta; \Omega}{\Gamma, x : \mathbf{1} \vdash \mathbf{recv}\ x\ (() \Rightarrow P) :: \Delta; \Omega}$$

$$[\bot R] \quad \frac{\Gamma \vdash P :: \Delta; \Omega}{\Gamma \vdash \mathbf{recv}\ x\ (() \Rightarrow P) :: x : \bot, \Delta; \Omega} \qquad [\bot L] \quad \frac{}{x : \bot \vdash \mathbf{send}\ x\ () :: \cdot; \Omega}$$

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

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

$$[\oplus R_k] \quad \frac{\Gamma \vdash P :: A_k; \Omega \qquad k \in L}{\Gamma \vdash \mathbf{send}\ x\ k; P :: x : \oplus\{l : A_l\}_{l \in L}; \Omega} \qquad [\oplus R] \quad \frac{\Gamma, x : A_l \vdash P_l :: \Delta; \Omega \qquad (\forall l \in L)}{\Gamma, x : \oplus\{l : A_l\}_{l \in L} \vdash \mathbf{recv}\ x\ (l \Rightarrow P_l)_{l \in L} :: \Delta; \Omega}$$

Figure 2.4: Static semantics of the type system

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 exception handler Ω. Such rules are also called the static semantics of the type system.

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. 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.

[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.

[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.