Exceptions in 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 [**Howard1980**]. The computational interpretation of linear logic, firstly given by Abramsky, lays the foundation for the connection between π-calculus and linear logic [**Abramsky1933**]. 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 [**Caires2014**].

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` [**Milner1997**]. 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 [**Mos2014**].

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

Below is a sequent proof in intuitionistic linear logic.

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

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 that uses all the channels in the antecedent and provides a channel in the succedent.

The cut is included as a primitive rule in the system. 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.

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

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 A_i's implies the disjunction of C_j's. 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 [**Wadler2012**]. 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.

$$\frac{\overset{P}{\vdash A, \Delta} \qquad \overset{Q}{\vdash A^\perp, \Delta'}}{\vdash \Delta, \Delta'}\ \text{cut}_A$$

Since the channel that connects process P and process Q are on both sides, classical logic blurs the distinction between input and output.

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

The annotated two-sided judgment can be viewed as a process that communicates along channels $x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m$, where each channel obeys its protocol requirements. It is possible to demonstrate the correctness of this system by converting it back to intuitionistic linear logic through double negation translation [**Friedman1978**]. 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
```

Secondly, static checking for the interpreter does not align well with implicit weakening. According to implicit rules, 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 channels unused.

Chapter 2

Type system

TBD

Appendix A

Proofs