Simply Typed Lambda Calculus with Channel

Tianyao Zhou (202101640)

December 29, 2021

1 Project Description

In this project, we formalize Simply Typed Lambda Calculus extended with asynchronized message-passing channels(referred as λ_{ch}below) and formulate and prove progress and preservation theorem on both the term level and the configuration level.

Elements of $\lambda_{ch}$is inspired by Simon John Fowler[1].

1.1 Syntax

$$
\begin{array}{rcl}
t & := & x \\
& | & \backslash x : T, t \\
& | & \mathsf{unit} \\
& | & t\ t \\
& | & \mathsf{let}\ x = t\ \mathsf{in}\ t \\
& | & \mathsf{fork}\ t \\
& | & \mathsf{give}\ c\ t \\
& | & \mathsf{take}\ c \\
& | & \mathsf{mkCh}\ T \\
& | & \mathsf{Ch}\ n\ T
\end{array}
$$

The first rule is variable x.

The second rule is abstraction on x of type T in term t.

The third rule is unit value.

The forth rule is let expression.

The fifth rule is fork expression, which fork a new thread.

The sixth rule is send expression, which sends t to a channel c.

The seventh rule is receive expression, which receives an value from a channel c.

The eighth rule is channel creation expression, which create a new channel for communication.

The ninth rule is intermidiate representation of a channel on data of type T and a natural number identification n.

1.2 Types

$$\begin{array}{rcl} T & := & \mathsf{Unit} \\ & | & T \to T \\ & | & \mathsf{Chan}\ T \end{array}$$

The first rule is Unit type with only one inhabitant unit.
The second rule is function type.
The third rule is channel type, which only carries data of type T.

1.2.1 Typing Rules

$$\frac{\Gamma[x] = T}{\Gamma \vdash x : T}$$

$$\frac{x : T, \Gamma \vdash t : T'}{\Gamma \vdash \backslash x : T, t : T \to T'}$$

$$\frac{}{\Gamma \vdash unit : Unit}$$

$$\frac{\Gamma \vdash t_1 : T \qquad \Gamma \vdash t_2 : T \to T'}{\Gamma \vdash t_1\ t_2 : T'}$$

$$\frac{\Gamma \vdash t_1 : T \qquad x : T, \Gamma \vdash t_2 : T'}{\Gamma \vdash \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 : T'}$$

$$\frac{\Gamma \vdash t : Unit \to T}{\Gamma \vdash \mathsf{fork}\ t : Unit}$$

$$\frac{\Gamma \vdash c : \mathsf{Chan}\ T \qquad \Gamma \vdash t : T}{\Gamma \vdash \mathsf{give}\ c\ t : Unit}$$

$$\frac{\Gamma \vdash c : \mathsf{Chan}\ T}{\Gamma \vdash \mathsf{take}\ c : T}$$

$$\frac{}{\Gamma \vdash \mathsf{mkCh}\ T : \mathsf{Chan}\ T}$$

$$\frac{}{\Gamma \vdash \mathsf{Ch}\ n\ T : \mathsf{Chan}\ T}$$

1.3 Configuration

The configuration consists of a list of channels and a list of threads. Each channel is a list of terms and each thread is a term that's either under evaluation or a value.

$$config := [c_0, \ldots, c_n]; [e_1, \ldots, e_m]$$

1.4 Operational Semantics

1.4.1 Evaluation Context

The evaluation context is defined below.

$$\begin{array}{rcl} E & := & \circ \\ & | & E\ t \\ & | & t\ E \\ & | & \mathsf{let}\ x = E\ \mathsf{in}\ t \\ & | & \mathsf{fork}\ E \\ & | & \mathsf{give}\ E\ t \\ & | & \mathsf{give}\ t\ E \\ & | & \mathsf{take}\ E \end{array}$$

And context filling rules.

$$\frac{}{e = \circ\{e\}}$$

$$\frac{e_1 = E\{e\}}{e_1\ e_2 = (E\ t)\{e\}}$$

$$\frac{value\ v_1 \qquad e_2 = E\{e\}}{v_1\ e_2 = (v_1\ E)\{e\}}$$

$$\frac{e_1 = E\{e\}}{\mathsf{let}\ x = e_1\ \mathsf{in}\ t = (\mathsf{let}\ x = E\ \mathsf{in}\ t)\{e\}}$$

$$\frac{e_1 = E\{e\}}{\mathsf{fork}\ e_1 = (\mathsf{fork}\ E)\{e\}}$$

$$\frac{e_1 = E\{e\}}{\mathsf{give}\ e_1\ e_2 = (\mathsf{give}\ E\ e_2)\{e\}}$$

$$\frac{value\ v_1 \qquad e_2 = E\{e\}}{\mathsf{give}\ v_1\ e_2 = (\mathsf{give}\ v_1\ E)\{e\}}$$

$$\frac{e_1 = E\{e\}}{\mathsf{take}\ e_1 = (\mathsf{take}\ E)\{e\}}$$

1.4.2 Term-level rules

$$\frac{value\ v}{\backslash x : T, t\ v \to t[x/v]}$$

$$\frac{value\ v}{\mathsf{let}\ x = v\ \mathsf{in}\ t \to t[x/v]}$$

$$\frac{e \to e'}{E\{e\} \to E\{e'\}}$$

1.4.3 Configuration-level rules

$$\frac{}{[c_0, \ldots, c_n]; [fork(\backslash x : \mathsf{Unit}, t), \ldots, e_m] \to [c_0, \ldots, c_n]; [\mathsf{unit}, \ldots, e_m, t]}$$

$$\frac{value\ v}{[c_0, \ldots, c_n]; [\mathsf{give}\ (\mathsf{Ch}\ p\ T)\ v, \ldots, e_m] \to [c_0, \ldots, c_p + [v], \ldots, c_n]; [\mathsf{unit}, \ldots, e_m]}$$

$$\frac{}{[c_0, \ldots, c_p = [v] + c'_p, \ldots, c_n]; [\mathsf{take}\ (\mathsf{Ch}\ p\ T), \ldots, e_m] \to [c_0, \ldots, c'_p, \ldots, c_n]; [v, \ldots, e_m]}$$

$$\frac{e \to e'}{[c_0, \ldots, c_n]; [E\{e\}, \ldots, e_m] \to [c_0, \ldots, c_n]; [E\{e'\}, \ldots, e_m]}$$

$$\frac{\neg value\ e \qquad [c_0, \ldots, c_n]; [e, \ldots, e_m] \to [c'_0, \ldots, c'_p]; [e', \ldots, e_q]}{[c_0, \ldots, c_n]; [E\{e\}, \ldots, e_m] \to [c'_0, \ldots, c'_p]; [E\{e'\}, \ldots, e_q]}$$

$$\frac{value\ v}{[c_0, \ldots, c_n]; [v, e_1, \ldots, e_m] \to [c_0, \ldots, c_n]; [e_1, \ldots, e_m]}$$

1.5 Progress and Preservation Theorem

Progress at term level is formulated as below.

Theorem 1 *For all closed term t, if t is of type T, then one of*

1. *t is a value*
2. *there exists a term t', such that $t \to t'$*
3. *there exists a evaluation context E, such one of*
   (a) *there exists a type T, such that $t = E\{\mathsf{mkCh}\ T\}$*
   (b) *there exists a value c, such that $t = E\{\mathsf{take}\ c\}$*
   (c) *there exists two values c and v, such that $t = E\{\mathsf{give}\ c\ v\}$*
   (d) *there exists a value f, such that $t = E\{\mathsf{fork}\ f\}$*

   holds

holds.

Preservation at term level is formulated as below.

Theorem 2 *For all closed term t of type T and t', if $t \to t'$, then t' is also of type T.*

Weak progress at configuration level is formulated as below.

Theorem 3 *For all configuration consists of a list of channels chs and a list of threads thrds, if* $thrds = t :: tail'$ *and* t *is a closed term of type* T*, and all channels refered in* t *is in chs, then one of*

1. t *is a value*
2. *there exists a term* t'*, a list of channels* chs' *and a list of threads* $thrds'$ *such that* $chs; thrds \to chs'; t' :: thrds'$
3. *there exists a program context* E*, a channel* $\mathsf{Ch}\ p\ T$*, such that* $t = E\{\mathsf{take}\ (\mathsf{Ch}\ p\ T)\}$ *and* $c_p = []$

Preservation theorem at configuration level is formulated as below.

Theorem 4 *For all configuration consists of a list of channels chs, two list of threads thrds and* $thrds'$ *and a closed term* t'*, if* $thrds = t :: tail'$ *and* t *is a closed term of type* T *and* t *is not a value, and all channels in chs are the typed according to references in* t*, and* $chs; thrds \to chs'; t' :: thrds'$ *then* t' *is also of type* T*.*

2 Problems

2.1 How to model the operational semantics of λ_{ch}?

There are different ways of writing small-step semantics. One is structural operational semantics, which captures the evaluation order of a certain type of expression by a series of rules that evaluates sub-expressions.

Another one is contextual operational semantics, which captures evaluation order by evaluation contexts.

When writing operational semantics, we can have a general case,

$$\frac{e \to e'}{E\{e\} \to E\{e'\}}$$

and base cases for expressions in the language.

They are equivalent. But the latter has some superficial advantages. It makes writing rules simpler and proof cleaner[2].

I chose to model $\lambda_{ch}$with contextual operational semantics, because it allows one organize proof in a modular manner and this is the method used in Fowler's paper[1].

2.2 How to define configuration?

In the original paper[1], the author used process calculi to represent evaluation configuration. However, it's merely a way to represent threads and bind channel identification to a buffer(a list). Besides in the paper[1], it also mentions canonical form of configuration, which is very similar to two lists, one of which represents channels and the other represents threads. So in this project, I modeled the configuration as two lists.

2.3 How to capture concurrency?

There are many ways to capture concurrency of programs. One is true concurrency, which allows multiple threads to progress in one step. Another one is interleaved execution, which nondeterministically select one thread to progress in one step.

However, real-world concurrent programs always comes with a scheduler, which only allows some threads to be executed at the same time and may or may not be biased. To make this project simpler, we only reduce the first expression in the thread list and append the forked thread to the end, though this is not concurrency in any sense. We could improve on this, but I don't know how to do this. What I have in mind is that similar to evaluation context of terms, we can have context for a list like $heads + [t] + tails$ saying that there are some elements before t represented by $heads$ and some elements after t represented by $tails$.

2.4 How to define contextual operational semantics in Coq?

The main issue here is how and where we describe a term to be a value in evaluation contexts.

We can describe it as an inductive type and embed the type in the term definition. When defining the structure of evaluation contexts, it naturally fits in. However, if one type of expression could both be a value and a reducible term, for example the tuple expression, it will appear both in the definition of values and terms. This seems a bit redundant.

We can also use a predicate to describe a term to be a value. It is more general than the former method, and allows a cleaner definition of terms.

In this project, I chose the latter way.

If we don't embed value in evaluation contexts' definitioin, we need another way to say the evaluation context is well-formed.

We can use a predicate for it. But we can also state it in context filling rules by saying that only when the context is well-formed, we can fill it.

When proving theorems, we usually do an induction on evaluation context. I feel the former method may allow simpler automation, while the latter requires inversion tactic to recover information.

In this project, I chose the latter way because $\lambda_{ch}$is a rather small language.

2.5 How to formulate progress and preservation theorem?

Because we have message-passing communication in λ_{ch}, receiving from an empty channel could cause evaluation to stuck and our type system is not powerful enough to guarantee progress property. The progress rule statement in *Programming Language Foundations* is not suitable here.

At the beginning, we planned to model the language with structural operational semantics. And Vlad sugguests that we could add a busy-waiting rule when evaluation gets stuck and state progress the same way as STLC.

$$\frac{c_p = []}{[c_0, \ldots, c_p, \ldots, c_n]; [\mathsf{take}\ (\mathsf{Ch}\ p\ T), e_1, \ldots, e_m] \to [c_0, \ldots, c_p, \ldots, c_n]; [\mathsf{take}\ (\mathsf{Ch}\ p\ T), e_1, \ldots, e_m]}$$

However, this rule seems a bit weird. Now that I've chosen the contextual operational, which allows us to pinpoint the next sub-epxression to be evaluated, we can state this as a special case in the progress theorem. Clearly this progress can only be a weak progress theorem, so why not show the "weakness" in the theorem definition by treating receiving from a empty channel a specail case instead of adding strange rules in operational semantics?

Besides, we can seperate progress theorem at term level and at configuration level by making all concurrency-related expression special cases of progress theorem at term level. This also allows cleaner organization of proofs because we can use progress theorem at term level when proving progress theorem at configuration level.

This is inspired by the paper[1] this project based on.

3 Experience with Coq

I really like the expressiveness of Coq. I can model inductively defined concepts define theorems easily. It also helps you keep track of hypotheses and goals when doing proofs, and it forces you to consider all the cases, so you don't make mistakes. When modeling complicated problems, it is easy for people to miss something in their proofs. However with Coq, either you modeled the problem in a wrong way or you get stuck when proving. There is no tricks here.

However, in Coq, it is really tempting to blindly try inversion, induction or other tactics. This is definitely not a good way of writing proofs, because if you modeled the problem erroneously, you can easily waste tens of minutes on it. The better way is to examine the hypotheses and goals carefully and see what you need to prove the goal and what is missing from the hypotheses. You need to understand the problem before proving it. In many situations, you will need to define some lemmas to help you. If you understand the problem, it is very natural to come up with one when you get stuck.

What I don't like Coq is that there are many house-keeping things you need to do when proving like duplicating hypotheses, generalizing terms and remember terms before induction. And the standard library is not as comfortable to use as Haskell's, possibly because it is not built around type classes.

References

[1] Simon John Fowler. "Typed concurrent functional programming with channels, actors and sessions". In: (2019).

[2] Robert Harper. *Practical foundations for programming languages*. Cambridge University Press, 2016.