

A Formalization of the Process Algebra CCS in HOL4

Chun Tian

Scuola di Scienze, Università di Bologna

`chun.tian@studio.unibo.it`

Numero di matricola: 0000735539

Abstract. An old formalization of the Process Algebra CCS (no value passing, with explicit relabeling operator) has been ported from HOL88 to HOL4 (Kananaskis-11 and later). Transitions between CCS processes are defined by SOS (Structured Operational Semantics) inference rules, then all algebraic laws (including the expansion theorem) were proved upon SOS transition rules.

We have used HOL4's new co-inductive relation support to re-define strong and weak bisimulation equivalences, and shows that the new definitions are equivalent with old ones. Finally, there's decision procedure for automatical detection of CCS transitions. The aim is to provide a up-to-date sound and effective tool to support verification and reasoning about process algebra specifications, and to provide a formal logic basis for further theoretical developments in Concurrent Theory.

1 Introduction

Concurrency Theory [1] has been successful for explaining many concurrency phenomenons in Computer Science, where Turing completeness is not enough. In this theory, reactive systems can be modelled as (possibly infinite) directed graphs of some (atomic) states with labeled edges as transitions between these states. Reactive systems defined in this way are called *Labeled Transition Systems* (LTSs) [2].

But it's usually inconvenient to describe and use (rooted) LTSs when studying the behaviors of reactive systems, especially when there're infinite states. To overcome such difficulties, different compact representations were invented as languages for describing reactive systems. One notable is Milner's *Calculus of Communicating Systems* (CCS) [3]. Such a language can be considered as independent process algebras (or syntactic calculus), with LTSs as its underlying semantic models. But process algebras like CCS contain more information than LTS: it has ability to synchronize pairs of actions, and all expressive powers come from synchronizations. The possible transitions of processes can be defined by a group of inference rules called Structural Operational Semantics (SOS). Then these rules can be used as a set of axioms for proving all of algebraic laws about the equivalence of CCS processes.

There're usually two methods for studying the behaviors of reactive systems. One is through the behavior equivalence checking between the *specification* and *implementation* of the same reactive system. Notable behavior equivalences include the strong and (rooted) weak bisimilarities. The other method is to do model checking directly on the implementation, in which the target properties were usually expressed in μ -Calculus or Hennessy-Milner Logics (HML). Both methods have their pros and cons.

Concurrency Theory is a proof-intense area, as shown in many textbooks its success was based on more trusted theories (and related theorems) than specific algorithms. However, most of these theorems were only done in pencil-and-papers, i.e. they have no formal proofs, nor there's currently any formal archieve for holding formal proofs in this area. When people was publishing new process algebras like new CCS variants, usually there's no corresponding software environments to actually play with the new theory, nor the theorems presented in the paper were formally verified. We think the main reason is the lack of the formal basis in which fundamental datatypes and relations and theorems were supplied for further extension.

The CCS formalization done by Monica Nesi in 1992-1995 in was a success, the work was ever a major success story for HOL theorem prover (HOL88 and HOL92 at that time), not to mention that both CCS and HOL (from Edinburgh LCF) are derived from the initial pioneer work

by the same scholar, Robin Milner. After more than 20 years, HOL88 and HOL92 were replaced by HOL4 (latest release is called Kananaskis-11), even the underlying programming language for writing HOL has changed from Classic ML (defined on top of Common Lisp) to Standard ML. But the CCS formalization seems being forgotten and to the best of our knowledge there's no other CCS formalizations done for HOL or any other theorem provers like Coq. As a result, new results in Concurrent Theory for the last 20 years were almost never formally verified. This is a rather unacceptable situation from the view of the author. As a little contributions to academic, the author has spent several months porting all the old code (provided by Prof. Nesi) to latest HOL4 with some improvements, new theorems, also with new features (e.g. co-inductive relation) used as alternative ways to define certain important concepts (e.g. bisimulation equivalence) in concurrency theory.

The work contains about 6000 lines of Standard ML code¹, with examples. About 170 theorems were proved, including the Expansion Theorem. There's also a ML function for computing CCS transitions from any given CCS process, and the output is a theorem which completely gave the characters for the transitions. Programs written in this way can be seen as kind of trusted computing, in the sense that, whenever the computation is terminated, the result must be correct, since it's a proven theorem. This prevented any doubts from the possible bugs existed in the software. For instance, if there's a hiding bug in Concurrency Workbench and have caused two complicated CCS processes being considered as equivalent but actually they're not, there's almost no way to know, except for comparing the results from another different software. Of course, any program including our ML functions could have bugs, thus we're not sure if the program can always give the result for any valid input, but as long as it does have a result, the result MUST be true (since it's a theorem) thus can be fully trusted. We think this is a major advantage to use theorem provers in place of normal programming languages for implementing important software verification tools as reliability is more important than any other factors.

2 Background

Back to one year ago (May 2016), soon after the author has just followed his 2nd-year course *MODELLI E SISTEMI CONCORRENTI (Concurrent Models and Systems)* of Computer Science (Informatica) in University of Bologna, he was looking on Internet for connections between CCS and the HOL theorem prover (HOL4) that he just began to learn. At that time, the author didn't know how to use HOL4 yet, but he liked it and the whole *formal methods* area very much.

Fortunately, the author found a paper [4] about the history of HOL theorem prover, in which Mike Gordon wrote:

I had been impressed by how the Expansion Theorem of Milner's Calculus of Communicating Systems (CCS) [5] enabled a direct description of the behaviour of a composite agent to be calculated from the parallel composition of its individual components.
... Incidentally, not only was CCS's Expansion Theorem an inspirational stepping stone from LCF via LSM to HOL, but recently things have come 'full circle' and Monica Nesi has used HOL to provide proof support for CCS, including the mechanisation of the Expansion Theorem [6].

This is how the author found the paper of Prof. Monica Nesi and some deep connections between LCF (Robin Milner), HOL (Mike Gordon) and CCS (again Robin Milner!). Then on May 24, 2016, the author posted on HOL's mailing list asking for the proof scripts code mentioned in the paper. Surprisingly Mike Gordon replied that mail with the following contents:

"Hi,
I don't have Monica's email address, but I do have her husband's, so I've forwarded your email to him.
Cheers,
Mike"

¹ Currently it's stored in GitHub: <https://github.com/binghe/informatica-public/tree/master/CCS>, we're trying to submit it into HOL's official examples library.

In later mails in the same day, Mike Gordon also told the author that, “Monica Nesi is also in Italy: she works at the University of L’Aquila (unless she has moved recently)”.²

Just one day later, on May 25, 2016, Professor Nesi sent an email to the author:

“Dear student,

I am Monica Nesi (a "she" :-)) from University of L’Aquila. My HOL scripts on CCS formalization are not available on Internet, but just give me time to find back my files and I will send them to you. I haven’t been working on that after my PhD, already more than 20 years ago, and I am pleased that someone might be interested in having a look and maybe do something similar in HOL4.

I have some deadlines to meet by Friday. I will come back to you asap.

Best regards, Monica Nesi”

And finally on June 7, the author received about 4500 lines of HOL88 proof scripts in 21 disk files. Here is a list of these files:

```
syntax.ml
syntax_aux.ml
aux_fun.ml
basic_rule_tac.ml
opsem.ml
runM.ml
StrongEQ/basic_conv_tac.ml
StrongEQ/basic_fun.ml
StrongEQ/par_strong_laws.ml
StrongEQ/parallel_new.ml
StrongEQ/rec_strong_laws.ml
StrongEQ/relab_strong_laws.ml
StrongEQ/restr_strong_laws.ml
StrongEQ/strong_par_conv_new.ml
StrongEQ/strong_rec_conv.ml
StrongEQ/strong_relab_conv.ml
StrongEQ/strong_restr_conv.ml
StrongEQ/strong_sem.ml
StrongEQ/strong_sum_conv.ml
StrongEQ/strong_tac.ml
StrongEQ/sum_strong_laws.ml
```

These include basic CCS definitions, proofs for all algebraic laws (including the expansion law) for strong equivalence, and a complicated ML function (in `runM.ml`) for automatically computing the transitions from a given CCS process.

The author saved these code and continued learning HOL4. Finally, starting from Jan 2017, the author was able to read proof scripts written in HOL4 and prove some new theorems in it. With several theorem proving projects doing in parallel, and kindly help from people of HOL community (Thomas Therk, Michael Norrish and many others), the author quickly improved his proof skills (in HOL4) and programing kills (in Standard ML). With a Classic ML document [7] found on Internet, porting Classic ML code into Standard ML seems quite straightforward.

The porting process of old HOL88 proof scripts is not very difficult: the underlying HOL logic didn’t change at all, so is the name of almost all tacticals and other ML functions. So basically what the author did is the following:

1. Copy a piece of ML code from old files into current HOL proof script;
2. Change the grammar from Classic ML to Standard ML;
3. Replay the proof in HOL4’s interactive proof manager;
4. Make necessary changes for those tacticals which has slightly changed their semantics in HOL4;
5. Save the working proof in forms of HOL `store_thm` function calls and go to next theorem.

² I should have really searched on Internet first...

Beside those small new inventions and some new ways to define old concepts (e.g. strong equivalence), the major efforts in this project were lots of time spent on carefully replaying each of the proofs: most proofs on strong laws were quite long (usually made of hundreds of tacticals, in four or five levels). In the time of HOL88, there's NO automatic first-order proof searching tools like Mason and Metis, nor the Q method, thus all proofs were done manually including each small steps involving only bool theory and higher order logic (beta-conversion, etc.). And whatever any literal terms are mentioned in the proof, full type info must be given manually, this makes the proof longer, but each step is very clear.

On the other side, Prof. Roberto Gorrieri was informed since the very beginning. He has encouraged the author to continue this project and use it as part of the exam of his course. Thus the current paper (as a project report) is well supported by Prof. Roberto and is actually the exam paper for the author's *MODELLI E SISTEMI CONCORRENTI (Concurrent Models and Systems)* course.

3 A Formalization of CCS in HOL4

The precise class of CCS we have formalized here, is CCS^{rel} , Finitary CCS with relabeling operator. Relabeling is necessary as part of the syntax of CCS processes not only because it's useful for defining a number of variants from any given pattern, but also the fact that, without expliciting relabeling operator there's no way to define linking operators which is very useful in many models.

3.1 Labels and Actions

In most literals, the concept *Action* was usually defined as the union of a countable set of input actions \mathcal{L} and output actions (co-actions) $\overline{\mathcal{L}}$ and a special invisible action $\tau \notin \mathcal{L} \cup \overline{\mathcal{L}}$. Actions are also called labels when LTS is considered.

In the formalization of CCS, however, it's better to have two distinct types: the type **Label** is the union of input and output (visible) actions, and the data type **Action** is the union of all visible and invisible actions. This is better because certain formulae and constructors in CCS doesn't accept τ as valid actions, e.g. the restriction operator of CCS. Thus, having two distinct types it's possible to make sure all CCS terms constructed from all possible values of their parameters are valid. Thus we have defined the following two data types in HOL:

```
Datatype 'Label = name string | coname string';
Datatype 'Action = tau | label Label';
```

Noticed that, the type **Action** contains the type **Label**, but they're not in sub-type relation (there's no such support in HOL): for operators accepting **Action**, if a term of **Label** were used, will result into type mismatch.

Here we have used the string type provided in HOL's **stringTheory**, thus the distinction and injectivity of the type **Label** also depends on the distinction and injectivity of type **string** proved in HOL's **stringTheory**. In theory, it's possible to use type variables instead of the string type and allow arbitrary types being used as labels, however we found no such needs so far.

Thus, a input action a must be represented as "**label (name "a")**" in HOL, while a output action (co-action) \bar{a} must be represented as "**label (coname "a")**". Instead, the invisible action can be written either in ASCII form **tau** or Greek letter τ in Unicode. This makes literal actions quite long, and we have defined the following syntactic sugars as compat representations of actions (not part of the original CCS work):

```
⊢ In act = label (name act)
⊢ Out act = label (coname act)
```

Above two definitions may be useful for final users who wants to express CCS processes as short as possible in HOL, but they were not used in any theorem that we have proved in this project. However, we have provided functions to easily transform between the original form and compact forms of actions for any CCS terms and theorems. They're part of the **CCSsims** theory:

```

val from_compact          : thm -> thm
val from_compact_tm      : term -> term
val to_compact           : thm -> thm
val to_compact_tm        : term -> term

```

The functions `from_compact` and `from_compact_tm` can be used to convert theorems and terms containing `In` and `Out` actions to the original (long) representations. And the functions `to_compact` and `to_compact_tm` convert the original representations to the compact (short) forms. For example, to convert between the normal form and compact form for the action \bar{a} , we can use functions `from_compact_tm` and `to_compact_tm` in the following ways:

```

> to_compact_tm ‘‘label (cname "a")‘‘;
val it = ‘‘Out "a"‘‘: term
> from_compact_tm ‘‘Out "a"‘‘;
val it =
  ‘‘label (cname "a")‘‘:
    term

```

The main operation on the types `Label` and `Action` is `COMPL` which gets their complements: (for convinence we also define the complement of τ as itself)

$$\begin{aligned}
&\vdash (\forall s. \text{COMPL } (\text{name } s) = \text{cname } s) \wedge \\
&\quad \forall s. \text{COMPL } (\text{cname } s) = \text{name } s \\
&\vdash (\forall l. \text{COMPL } (\text{label } l) = \text{label } (\text{COMPL } l)) \wedge (\text{COMPL } \tau = \tau)
\end{aligned}$$

As we know `Label` and `Action` are different types, the `COMPL` operator on them are actually overloaded operator of `COMPL_LAB` and `COMPL_ACT`, the complement operator for `Label` and `Action`.

The key theorem about `Label` says that, doing complements twice for the same label gets the label itself:

`COMPL_COMPL_LAB`:

$$\vdash \text{COMPL } (\text{COMPL } l) = l$$

There's also a similar theorem for the double-complements of `Action`.

The following table listed the notation of various actions, with notations from *Currency Workbench* [8] compared:

Action	notation	CWB	HOL (ASCII)	HOL (compact form)
internal action	τ	tau	tau	τ
input action	a	a	label (name "a")	In "a"
output action	\bar{a}	'a	label (cname "a")	Out "a"

3.2 Relabeling

In most literals, Relabeling is usually defined as a n unary substitution operator: $_ [b/a]$ takes a unary substitution b/a (hence, $a \neq b$), and a process p to construct a new process $p[b/a]$, whose semantics is that of p , where action $a(\bar{a})$ is turned into $b(\bar{b})$. And multi-label relabeling can be done by appending more unary substitution operators to the new process. The order of multiple relabelings is important, when new labels introduced in previous relabeling operation were further relabeled by later relabeling operations.

In our formalization, instead we support multi-label relabeling in one operation, and instead of using a list of substituions, we have defined a new fundamental type in called `Relabeling`.³ A `Relabeling` is a abstract type which is bijected into a subset of function of type `Label -> Label`, which is called the *representation* of the type `Relabeling`. Not all functions of type `Label -> Label` are valid representations of `Relabeling`, but only functions which satisfy the following property:

$$\vdash \text{Is_Relabeling } f \iff \forall s. f (\text{cname } s) = \text{COMPL } (f (\text{name } s))$$

³ The credits of this idea belongs completely to Prof. Monica Nesi. The author did nothing but the porting work from HOL88 to HOL4.

Noticed that, any identify function of type `Label -> Label` also satisfy above property. Thus, beside specific substitutions that we want, all relabeling functions must be able to handle all other labels too (just return the same label as input). (As we'll see later, such requirements could reduce the two rules for relabelling into just one).

But usually it's more convenient to represent relabeling functions as a list of substitutions of type `(Label × Label) list`. The operator `RELAB` can be used to define such a relabeling function. For instance, the term `RELAB [(name "b",name "a"); (name "d",name "c")]` can be used in place of a relabeling operator $[b/a, d/c]$, because its type is `Relabeling`. And it must be understood that, all relabeling functions are total functions: for all other labels except `a` and `c`, the substitution will be themselves (another way to express "no relabeling").

Finally, have the relabeling facility defined as a multi-label relabeling function and part of CCS syntax, we can completely avoid the complexity of the Syntactic Substitution (c.f. p.171 of [1]) which has a quite complicated recursive definition and seriously depends on some other recursive functions like $fn(\cdot)$ (free names) and $bn(\cdot)$ (bound names) for CCS processes (in our project, these functions are not included nor needed).

3.3 CCS processes and operators

The type `CCS` is defined as an inductive data type: (thus it must be finitary)

```
val _ = Datatype 'CCS = nil
      | var string
      | prefix Action CCS
      | sum CCS CCS
      | par CCS CCS
      | restr (Label set) CCS
      | relab CCS Relabeling
      | rec string CCS;
```

In HOL4, we have added some minimal grammar support, to represent CCS processes in more compact forms (not available in HOL88). The following table listed the notation of typical CCS processes and major operators supported by above definition, with notations from *Currency Workbench* [8] compared:

op name	notation	CWB	HOL (ASCII)	HOL (compact)
Deadlock (nil)	0	0	nil	nil
Prefix	$a.0$	a.0	prefix (label (name "a")) nil	In "a"..nil
Sum	$p + q$	p + q	sum p q	p + q
Parallel	$p q$	p q	par p q	p q
Restriction of action	$(\nu a)p$	p \a	nu { "a"} p	ν "a" p
Restriction of actions	$(\nu L)p$	p \L	restr L p	ν L p

For Relabeling, as we described in last section, to express $p[b/a]$, it must be written as `relab p (RELAB [(name "b",name "a")])`, which is a little long.

For CCS process defined in one or more constants, in our formalization in HOL4, all constants must be written into single term. (This is necessary for theorem proving, because otherwise there's no way to store all information into single variable in CCS-related theorems) The syntax for defining new constants is `rec` and the syntax to actually use a constant is `var`. To see how these operators are actually used, consider the following CCS process (the famous coffee machine model from [1]):

$$\begin{aligned}
VM &\stackrel{def}{=} coin.(ask-esp.VM_1 + ask-am.VM_2) \\
VM_1 &\stackrel{def}{=} \overline{esp-coffee}.VM \\
VM_2 &\stackrel{def}{=} \overline{am-coffee}.VM
\end{aligned}$$

In our formalization in HOL4, above CCS process can be represented as the following single term:

```

‘‘rec "VM"
  (In "coin"
    ..
    (In "ask-esp" .. (rec "VM1" (Out "esp-coffee"..var "VM"))) +
    In "ask-am" .. (rec "VM2" (Out "am-coffee"..var "VM"))))‘‘

```

That is, for the first time a new constant appears, use **rec** with the name of constants as string to “declare” it; when any constant appears again, use **var** to access it.

Finally, although not part of the formal definition, the **if-then-else** construct from value-passing CCS is naturally supported by HOL. This is because, for any boolean value b and two terms t_1 and t_2 of type α , the term **if** b **then** t_1 **else** t_2 has also the type α . Thus the conditional term can legally appears inside other CCS processes as a sub-process. We’ll see in next section that it’s necessary for handling transitions of CCS processes containing constants.

3.4 CCS transitions and SOS inference rules

The transitions of CCS processes were defined by the following Structural Operational Semantics (SOS for short) rules:

$$\begin{array}{ll}
(\text{Perf}) \frac{}{\mu.p \xrightarrow{\mu} p} & (\text{Par}_1) \frac{p \xrightarrow{\mu} p'}{p|q \xrightarrow{\mu} p'|q} \\
(\text{Rec}) \frac{p \xrightarrow{\mu} p'}{C \xrightarrow{\mu} p'} C \stackrel{\text{def}}{=} p & (\text{Par}_2) \frac{q \xrightarrow{\mu} q'}{p|q \xrightarrow{\mu} p|q'} \\
(\text{Sum}_1) \frac{p \xrightarrow{\mu} p'}{p+q \xrightarrow{\mu} p'} & (\text{Par}_3) \frac{p \xrightarrow{\alpha} p' \quad q \xrightarrow{\bar{\alpha}} q'}{p|q \xrightarrow{\tau} p'|q'} \\
(\text{Sum}_2) \frac{q \xrightarrow{\mu} q'}{p+q \xrightarrow{\mu} q'} & (\text{Res}) \frac{p \xrightarrow{\mu} p'}{(\nu a)p \xrightarrow{\mu} (\nu a)p'} \mu \neq a, \bar{a}
\end{array}$$

Besides, we have two rules from CCS^{rel}:

$$\begin{array}{l}
(\text{Rel_1}) \frac{p \xrightarrow{\mu} p'}{p[b/a] \xrightarrow{\mu} (p'[b/a])} \mu \neq a \wedge \mu \neq \bar{a} \\
(\text{Rel_2}) \frac{p \xrightarrow{\alpha} p'}{p[b/a] \xrightarrow{\beta} (p'[b/a])} (\alpha = a \wedge \beta = b) \vee (\alpha = \bar{a} \wedge \beta = \bar{b})
\end{array}$$

In some literals, the rule Par₃ is also called “Com” (rule of communication), and the rule “Rec” is also called “Cons” (rule of constants). (Here we have preserved the rule names in the HOL88 work, because it’s easier to locate for their names in the proof scripts.)

From the view of theorem prover (or just first-order logic), these inference rules are nothing but an *inductive definition* on 3-ary relation **TRANS** (with compact representation $-() \rightarrow$) of type **CCS** \rightarrow **Action** \rightarrow **CCS** \rightarrow **bool**, generated by HOL4’s function **Hol_reln** [9]. Then we break them into separated theorems as primitive inference rules⁴:

```

PREFIX:  ⊢ u..E --u-> E
REC:     ⊢ CCS_Subst E (rec X E) X --u-> E1 ⇒ rec X E --u-> E1
SUM1:    ⊢ E --u-> E1 ⇒ E + E' --u-> E1
SUM2:    ⊢ E --u-> E1 ⇒ E' + E --u-> E1
PAR1:    ⊢ E --u-> E1 ⇒ E || E' --u-> E1 || E'

```

⁴ They’re considered as the axioms in our logic system, however they’re not defined directly as axioms.

HOL makes sure in such cases the logic system is still consistent.

PAR2: $\vdash E \text{ --}u\text{--} \Rightarrow E' \mid \mid E \text{ --}u\text{--} \Rightarrow E' \mid \mid E_1$
PAR3:
 $\vdash E \text{ --label } l\text{--} \Rightarrow E_1 \wedge E' \text{ --label } (\text{COMPL } l)\text{--} \Rightarrow E_2 \Rightarrow$
 $E \mid \mid E' \text{ --}\tau\text{--} \Rightarrow E_1 \mid \mid E_2$
RESTR:
 $\vdash E \text{ --}u\text{--} \Rightarrow E' \wedge$
 $((u = \tau) \vee (u = \text{label } l) \wedge l \notin L \wedge \text{COMPL } l \notin L) \Rightarrow$
 $\nu L E \text{ --}u\text{--} \Rightarrow \nu L E'$
RELAB:
 $\vdash E \text{ --}u\text{--} \Rightarrow E' \Rightarrow \text{relab } E \text{ rf --relabel rf } u\text{--} \Rightarrow \text{relab } E' \text{ rf}$

Noticed that, the two rules for Relabeling are merged into a single rule RELAB which uses a general relabeling function. In practice, to cover both Rel_1 and Rel_2, the relabeling function should be at least an identify function and this is guaranteed by the definition of relabeling functions as total functions.

Also noticed that, in the rule REC, a recursive function CCS_Subst was used. It has the following definition which depends on the conditional clause (if .. then .. else ..):

$\vdash (\forall E' X. \text{CCS_Subst nil } E' X = \text{nil}) \wedge$
 $(\forall u E' X. \text{CCS_Subst } (u..E) E' X = u..\text{CCS_Subst } E E' X) \wedge$
 $(\forall E_1 E_2 E' X.$
 $\quad \text{CCS_Subst } (E_1 + E_2) E' X =$
 $\quad \text{CCS_Subst } E_1 E' X + \text{CCS_Subst } E_2 E' X) \wedge$
 $(\forall E_1 E_2 E' X.$
 $\quad \text{CCS_Subst } (E_1 \mid \mid E_2) E' X =$
 $\quad \text{CCS_Subst } E_1 E' X \mid \mid \text{CCS_Subst } E_2 E' X) \wedge$
 $(\forall L E E' X.$
 $\quad \text{CCS_Subst } (\nu L E) E' X = \nu L (\text{CCS_Subst } E E' X)) \wedge$
 $(\forall E f E' X.$
 $\quad \text{CCS_Subst } (\text{relab } E f) E' X =$
 $\quad \text{relab } (\text{CCS_Subst } E E' X) f) \wedge$
 $(\forall Y E' X.$
 $\quad \text{CCS_Subst } (\text{var } Y) E' X = \text{if } Y = X \text{ then } E' \text{ else var } Y) \wedge$
 $\forall Y E E' X.$
 $\quad \text{CCS_Subst } (\text{rec } Y E) E' X =$
 $\quad \text{if } Y = X \text{ then rec } Y E \text{ else rec } Y (\text{CCS_Subst } E E' X)$

In HOL4, any inductive relation defined by command **Hol_reln** will return with three (well, actually four) theorems: 1) the rules, 2) the induction (and strong induction) theorem and 3) the “cases” theorem. Only with all these theorems, the relation can be precisely defined. For example, to prove certain CCS transitions are impossible, the following long “cases” theorem (which asserts that the relation is a fixed point) must be used:

$\vdash a_0 \text{ --}a_1\text{--} \Rightarrow a_2 \iff$
 $(a_0 = a_1..a_2) \vee (\exists E E'. (a_0 = E + E') \wedge E \text{ --}a_1\text{--} \Rightarrow a_2) \vee$
 $(\exists E E'. (a_0 = E' + E) \wedge E \text{ --}a_1\text{--} \Rightarrow a_2) \vee$
 $(\exists E E_1 E'. (a_0 = E \mid \mid E') \wedge (a_2 = E_1 \mid \mid E') \wedge E \text{ --}a_1\text{--} \Rightarrow E_1) \vee$
 $(\exists E E_1 E'. (a_0 = E' \mid \mid E) \wedge (a_2 = E' \mid \mid E_1) \wedge E \text{ --}a_1\text{--} \Rightarrow E_1) \vee$
 $(\exists E l E_1 E' E_2.$
 $\quad (a_0 = E \mid \mid E') \wedge (a_1 = \tau) \wedge (a_2 = E_1 \mid \mid E_2) \wedge$
 $\quad E \text{ --label } l\text{--} \Rightarrow E_1 \wedge E' \text{ --label } (\text{COMPL } l)\text{--} \Rightarrow E_2) \vee$
 $(\exists E E' l L.$
 $\quad (a_0 = \nu L E) \wedge (a_2 = \nu L E') \wedge E \text{ --}a_1\text{--} \Rightarrow E' \wedge$
 $\quad ((a_1 = \tau) \vee (a_1 = \text{label } l) \wedge l \notin L \wedge \text{COMPL } l \notin L)) \vee$
 $(\exists E u E' \text{rf}.$
 $\quad (a_0 = \text{relab } E \text{ rf}) \wedge (a_1 = \text{relabel rf } u) \wedge$
 $\quad (a_2 = \text{relab } E' \text{ rf}) \wedge E \text{ --}u\text{--} \Rightarrow E') \vee$
 $\exists E X. (a_0 = \text{rec } X E) \wedge \text{CCS_Subst } E (\text{rec } X E) X \text{ --}a_1\text{--} \Rightarrow a_2$

Here are some results proved using above “cases” theorem (i. e. they cannot be proved with only the SOS inference rules):

NIL_NO_TRANS:

$\vdash \neg(\text{nil} \text{ --}u\text{--} \rightarrow E)$

TRANS_IMP_NO_NIL:

$\vdash E \text{ --}u\text{--} \rightarrow E' \Rightarrow E \neq \text{nil}$

TRANS_SUM_EQ:

$\vdash E + E' \text{ --}u\text{--} \rightarrow E'' \iff E \text{ --}u\text{--} \rightarrow E'' \vee E' \text{ --}u\text{--} \rightarrow E''$

TRANS_PAR_EQ:

$\vdash E \parallel E' \text{ --}u\text{--} \rightarrow E'' \iff$
 $(\exists E_1. (E'' = E_1 \parallel E') \wedge E \text{ --}u\text{--} \rightarrow E_1) \vee$
 $(\exists E_1. (E'' = E \parallel E_1) \wedge E' \text{ --}u\text{--} \rightarrow E_1) \vee$
 $\exists E_1 E_2 l.$
 $(u = \tau) \wedge (E'' = E_1 \parallel E_2) \wedge E \text{ --label } l\text{--} \rightarrow E_1 \wedge$
 $E' \text{ --label } (\text{COMPL } l)\text{--} \rightarrow E_2$

TRANS_RESTRE_EQ:

$\vdash \nu L E \text{ --}u\text{--} \rightarrow E' \iff$
 $\exists E'' l.$
 $(E' = \nu L E'') \wedge E \text{ --}u\text{--} \rightarrow E'' \wedge$
 $((u = \tau) \vee (u = \text{label } l) \wedge l \notin L \wedge \text{COMPL } l \notin L)$

Finally, it's worth to mention that, the following induction theorem generated by `Hol_reln` was never used (nor needed) in this project:

TRANS_ind:

$\vdash (\forall E u. \text{TRANS}' (u..E) u E) \wedge$
 $(\forall E u E_1 E'. \text{TRANS}' E u E_1 \Rightarrow \text{TRANS}' (E + E') u E_1) \wedge$
 $(\forall E u E_1 E'. \text{TRANS}' E u E_1 \Rightarrow \text{TRANS}' (E' + E) u E_1) \wedge$
 $(\forall E u E_1 E'.$
 $\text{TRANS}' E u E_1 \Rightarrow \text{TRANS}' (E \parallel E') u (E_1 \parallel E')) \wedge$
 $(\forall E u E_1 E'.$
 $\text{TRANS}' E u E_1 \Rightarrow \text{TRANS}' (E' \parallel E) u (E' \parallel E_1)) \wedge$
 $(\forall E l E_1 E' E_2.$
 $\text{TRANS}' E (\text{label } l) E_1 \wedge \text{TRANS}' E' (\text{label } (\text{COMPL } l)) E_2 \Rightarrow$
 $\text{TRANS}' (E \parallel E') \tau (E_1 \parallel E_2)) \wedge$
 $(\forall E u E' l L.$
 $\text{TRANS}' E u E' \wedge$
 $((u = \tau) \vee (u = \text{label } l) \wedge l \notin L \wedge \text{COMPL } l \notin L) \Rightarrow$
 $\text{TRANS}' (\nu L E) u (\nu L E')) \wedge$
 $(\forall E u E' rf.$
 $\text{TRANS}' E u E' \Rightarrow$
 $\text{TRANS}' (\text{relab } E \text{ rf}) (\text{relabel } rf \text{ } u) (\text{relab } E' \text{ rf})) \wedge$
 $(\forall E u X E_1.$
 $\text{TRANS}' (\text{CCS_Subst } E (\text{rec } X E) X) u E_1 \Rightarrow$
 $\text{TRANS}' (\text{rec } X E) u E_1) \Rightarrow$
 $\forall a_0 a_1 a_2. a_0 \text{ --}a_1\text{--} \rightarrow a_2 \Rightarrow \text{TRANS}' a_0 a_1 a_2$

The purpose of above induction theorem is to assert the transition relation to the least fixed point of the function generated from SOS inference rules. On the other side, if we define an co-inductive relation from the same SOS rules, we get the same rules and “cases” theorems, and the only difference is another co-induction theorem in place of above induction theorem. This seems indicating that, the least fixed point coincides with greatest fixed point for Finitary CCS. This result is never formally proved, but both Prof. Gorrieri and the author believe it's true.

Also, we have noticed that, to prove certain CCS transition is impossible, it's enough to use just the above “cases” theorem. Since the CCS datatype itself is inductively defined, therefore

already Finitary, all those invalid transitions seems must be outside of the fixed point, in another word, they're even outside of the greatest fixed point.⁵

3.5 Decision procedure for CCS transitions

It's possible to use SOS inference rules and theorems derived from them for proving theorems about the transitions between any two CCS processes. However, what's more useful is to decision procedure which automatically decide all possible transitions and formally prove it.

For any CCS process, there is a decision procedure as recursive function, which can completely decide all its possible (one-step) transitions. In HOL, this decision procedure can be implemented as a normal Standard ML function `CCS_TRANS_CONV` of type `term -> theorem`, the returned theorem fully characterize the possible transitions of the input CCS process.

For instance, we know that the process $(a.0|\bar{a}.0)$ have three possible transitions:

1. $(a.0|\bar{a}.0) \xrightarrow{a} (0|\bar{a}.0);$
2. $(a.0|\bar{a}.0) \xrightarrow{\bar{a}} (a.0|0);$
3. $(a.0|\bar{a}.0) \xrightarrow{\tau} (0|0).$

To completely decide all possible transition, if done manually, the following work should be done:

1. Prove there exists transitions from $(a.0|\bar{a}.0)$ (optionally);
2. Prove each of above three transitions using SOS inference rules;
3. Prove there's no other transitions, using the "cases" theorems generated from the `TRANS` relation.

Here are the related theorems manually proved:

```
r1_has_trans:
  ⊢ ∃ l G.
    label (name "a")..nil || label (coname "a")..nil --l-> G
r1_trans_1:
  ⊢ label (name "a")..nil || label (coname "a")..nil
    --label (name "a")->
    nil || label (coname "a")..nil
r1_trans_2:
  ⊢ label (name "a")..nil || label (coname "a")..nil
    --label (coname "a")->
    label (name "a")..nil || nil
r1_trans_3:
  ⊢ label (name "a")..nil || label (coname "a")..nil
    --τ->
    nil || nil
r1_has_no_other_trans:
  ⊢ ¬∃ l G.
    ¬((G = nil || label (coname "a")..nil) ∧
      (l = label (name "a"))) ∨
      (G = label (name "a")..nil || nil) ∧
      (l = label (coname "a"))) ∨
      (G = nil || nil) ∧ (l = τ) ∧
      label (name "a")..nil || label (coname "a")..nil --l-> G
```

Instead, if we use the function `CCS_TRANS_CONV` with the root process:

```
> CCS_TRANS_CONV
  'par (prefix (label (name "a"))) nil)
    (prefix (label (coname "a"))) nil)'
```

⁵ If one day we had changed the definition of CCS datatype to allow infinite sums and parallels, the SOS inference rules should still work, but the transition relation should be then defined co-inductively, to allow valid transitions for both finitary and infinitary CCS processes.

As the result, the following theorem is returned:

```
ex_A: ⊢ label (name "a")..nil || label (coname "a")..nil --u-> E ⇔
  ((u = label (name "a")) ∧
   (E = nil || label (coname "a")..nil) ∨
   (u = label (coname "a")) ∧
   (E = label (name "a")..nil || nil)) ∨
  (u = τ) ∧ (E = nil || nil)
```

From this theorem, we can see there're only three possible transitions and there's no others. Therefore it contains all information expressed by previous manually proved 5 theorems (in theory we can also try to manually prove this single theorem, but it's not easy since the steps required will be at least the sum of all previous proofs).

As a further example, if we put a restriction on label "a" and check the process $(\nu a)(a.0|\bar{a}.0)$ instead, there will be only one possible transition:

```
ex_B: ⊢ ν {name "a"}
  (label (name "a")..nil || label (coname "a")..nil)
  --u->
  E ⇔ (u = τ) ∧ (E = ν {name "a"} (nil || nil))
```

The function `CCS_TRANS_CONV` has also a variant `CCS_TRANS_CONV'`, which outputs theorems in compact forms of actions.

It's possible to extract a list of possible transitions together with the actions, into a list. This work can be done automatically by the function `strip_trans`. Finally, if both the theorem and the list of transitions are needed, the function `CCS_TRANS` and its compact-form variant `CCS_TRANS'` can be used. For the previous example process $(a.0|\bar{a}.0)$, calling `CCS_TRANS'` on it in HOL's interactive environment has the following results:

```
> CCS_TRANS' 'In "a"..nil || Out "a"..nil';
val it =
  (|- !u E.
    In "a"..nil || Out "a"..nil --u-> E <=>
    ((u = In "a") /\ (E = nil || Out "a"..nil) \/
     (u = Out "a") /\ (E = In "a"..nil || nil)) \/
    (u = tau) /\ (E = nil || nil),
   [( 'In "a" ',
     'nil || Out "a"..nil ',
     'Out "a" ',
     'In "a"..nil || nil ',
     't ',
     'nil || nil ' ]):
  thm * (term * term) list
```

The main function `CCS_TRANS_CONV` is implemented in about 500 lines of Standard ML code, and it depends on many customized tacticals, and functions to access the internal structure of CCS related theorem and terms. We have tried our best to make sure the correctness of this function, but certain bugs are still inevitable. However, since it's implemented in theorem prover, and the return value of this function is a theorem, what we can guarantee is the following fact:

Whenever the function terminates with a theorem returned, the result can be fully trusted, and the CCS transitions indicated in the returned theorem is indeed all possible transitions from the input process. No matter if there're still other bugs in the program.

In another words, any remain bug in the program can only stop the whole function for returning a result, but as long as the result is returned, it cannot be wrong (i.e. a fake theorem). This sounds like a different kind of trusted computing than normal sense. In general, for any algorithm implemented in any normal programming languages, since the output is just a primitive value or data structure which can be arbitrary constructed or changed due to potential bugs in the implementation, the only way to trust these results, is to have the entire program carefully modelled and verified. But in our case, the Standard ML program code is not verified, but the result (once appears) can still be fully trusted, isn't this amazing?

4 Strong bisimulation, strong equivalence and co-induction

The concept of *Bisimulation* (and *Bisimulation Equivalence* with variants) stands at the central position of Concurrency Theory, as one major approach of model checking is to check the bisimulation equivalence between the specification and implementation of the same reactive system. Besides, It's well known that, Strong Equivalence as a relation, must be defined *co-inductively*. (And in fact, strong equivalence is one of the most well-studied co-inductive relation in computer science. [10]) In this section, we study the definition of strong and weak bisimulation and (bisimulation) equivalences, and their possible formalizations in HOL.

Recall the standard definition of strong bisimulation and strong equivalence (c.f. p.43 of [1]):

Definition 1. (Strong) bisimulation and (strong) bisimulation equivalent Let $TS = (Q, A, \rightarrow)$ be a transition system. A bisimulation is a relation $R \subset Q \times Q$ such that R and its inverse R^{-1} are both simulation relations. More explicitly, a bisimulation is a relation R such that if $(q_1, q_2) \in R$ then for all $\mu \in A$

- $\forall q'_1$ such that $q_1 \xrightarrow{\mu} q'_1, \exists q'_2$ such that $q_2 \xrightarrow{\mu} q'_2$ and $(q'_1, q'_2) \in R$,
- $\forall q'_2$ such that $q_2 \xrightarrow{\mu} q'_2, \exists q'_1$ such that $q_1 \xrightarrow{\mu} q'_1$ and $(q'_1, q'_2) \in R$.

Two states q and q' are bisimilar (or bisimulation equivalent), denoted $q \sim q'$, if there exists a bisimulation R such that $(q, q') \in R$.

Noticed that, although above definition is expressed in LTS, it's also applicable to CCS in which each process has the semantic model as a rooted LTS. Given the fact that, all states involved in above definition are target states of direct or indirect transition of the initial pair of states, above definition can be directly used for CCS.

In HOL88, there's no way to define co-inductive relation directly. However, it's possible to follow above definition literally and define bisimulation first, then define the bisimulation equivalence on top of bisimulation. Here are the definitions translated from HOL88 to HOL4:

```

⊢ STRONG_BISIM Bsm ⟷
  ∀ E E'.
    Bsm E E' ⇒
      ∀ u.
        (∀ E1. E --u-> E1 ⇒ ∃ E2. E' --u-> E2 ∧ Bsm E1 E2) ∧
        (∀ E2. E' --u-> E2 ⇒ ∃ E1. E --u-> E1 ∧ Bsm E1 E2)
⊢ E ~ E' ⟷ ∃ Bsm. Bsm E E' ∧ STRONG_BISIM Bsm

```

From the second definition, we can see that, $q \sim q'$ if there exists a bisimulation containing the pair (q, q') . This means that \sim is the union of all bisimulations, i.e.,

$$\sim = \bigcup \{R \subset Q \times Q : R \text{ is a bisimulation}\}.$$

In HOL4, the last formula can be proved with the notion “bigunion” in HOL’s `pred_setTheory` used. The only thing needed from CCS is above definition of strong equivalence (the definition of strong bisimulation is not needed at all):

```

⊢ STRONG_EQUIV = CURRY (BIGUNION {UNCURRY R | STRONG_BISIM R})

```

However, this theorem is not very useful for proving other results. And the use of `CURRY` and `UNCURRY` is to transform the relation from types between `CCS -> CCS -> bool` and `CCS × CCS -> bool`, since relations in HOL cannot be treated directly as mathematical sets.

The other way to define strong bisimulation equivalence is through the fixed point of the following function F : (c.f. p.72 of [1])

Definition 2. Given an LTS (Q, A, \rightarrow) , the function $F: \wp(Q \times Q) \rightarrow \wp(Q \times Q)$ (i.e., a transformer of binary relations over Q) is defined as follows. If $R \subset Q \times Q$, then $(q_1, q_2) \in F(R)$ if and only if for all $\mu \in A$

- $\forall q'_1$ such that $q_1 \xrightarrow{\mu} q'_1, \exists q'_2$ such that $q_2 \xrightarrow{\mu} q'_2$ and $(q'_1, q'_2) \in R$,

– $\forall q'_2$ such that $q_2 \xrightarrow{\mu} q'_2, \exists q'_1$ such that $q_1 \xrightarrow{\mu} q'_1$ and $(q'_1, q'_2) \in R$.

And we can see by comparing the definition of above function and the definition of bisimulation that (no formal proofs):

1. The function F is monotone, i.e. if $R_1 \subset R_2$ then $F(R_1) \subset F(R_2)$.
2. A relation $R \subset Q \times Q$ is a bisimulation if and only if $R \subset F(R)$.

Then according to Knaster-Tarski Fixed Point theorem, strong bisimilarity \sim is the greatest fixed point of F . And this is also the definition of co-inductive relation defined by the same rules.

In HOL4, since the release Kananaskis-11, there's a new facility for defining co-inductive relation. The entry command is `Hol_coreln`, which has the same syntax as `Hol_reln` for defining inductive relations. Using `Hol_coreln`, it's possible to define the bisimulation equivalence *directly* in this way: (here we has chosen a new relation name `STRONG_EQ`)⁶

```
val (STRONG_EQ_rules, STRONG_EQ_coind, STRONG_EQ_cases) = Hol_coreln ‘
  (!E E'.
    (!u.
      (!E1. TRANS E u E1 ==>
        (?E2. TRANS E' u E2 /\ STRONG_EQ E1 E2)) /\
      (!E2. TRANS E' u E2 ==>
        (?E1. TRANS E u E1 /\ STRONG_EQ E1 E2))) ==> STRONG_EQ E E')’;
```

HOL automatically generated 3 theorems from above definition:

`STRONG_EQ_rules:`

```
⊢ (∀ u.
  (∀ E1. E --u-> E1 ⇒ ∃ E2. E' --u-> E2 ∧ STRONG_EQ E1 E2) ∧
  ∀ E2. E' --u-> E2 ⇒ ∃ E1. E --u-> E1 ∧ STRONG_EQ E1 E2) ⇒
  STRONG_EQ E E')
```

`STRONG_EQ_coind:`

```
⊢ (∀ a0 a1.
  STRONG_EQ' a0 a1 ⇒
  ∀ u.
    (∀ E1.
      a0 --u-> E1 ⇒ ∃ E2. a1 --u-> E2 ∧ STRONG_EQ' E1 E2) ∧
    ∀ E2.
      a1 --u-> E2 ⇒ ∃ E1. a0 --u-> E1 ∧ STRONG_EQ' E1 E2) ⇒
  ∀ a0 a1. STRONG_EQ' a0 a1 ⇒ STRONG_EQ a0 a1)
```

`STRONG_EQ_cases:`

```
⊢ STRONG_EQ a0 a1 ⇔
  ∀ u.
    (∀ E1. a0 --u-> E1 ⇒ ∃ E2. a1 --u-> E2 ∧ STRONG_EQ E1 E2) ∧
    ∀ E2. a1 --u-> E2 ⇒ ∃ E1. a0 --u-> E1 ∧ STRONG_EQ E1 E2)
```

The first theorem is the original rules appearing in the definition. Roughly speaking, it's kind of rules for building a bisimulation relation in forward way, however this is impossible because of the lack of base rules (which exists in most inductive relation). And it's not original in this case, since it can be derived from the last theorem `STRONG_EQ_cases`.

The second theorem is the co-induction principle, it says, for what ever relation which satisfy the rules, that relation must be contained in strong equivalence. In another word, it make sure the target relation is the maximal relation containing all others.

The purpose of the last theorem (also called “cases” theorem), is to make sure the target relation is indeed a fixed point of the function F built by the given rules. However, it doesn't give any information about the size of such a fixed point. In general, if the greatest fixed point and least fixed point doesn't coincide, without the restriction by co-induction theorem, the rest two theorems will not give a precise definition for that relation. For strong equivalence, we already

⁶ Whenever ASCII-based HOL proof scripts were directly pasted, please understand the letter “!” as \forall , and “?” as \exists . They're part of HOL's term syntax. [11]

know that, the least fixed point of F is empty relation \emptyset , and the great fixed point is the strong equivalence \sim . And in fact, the “cases” theorem has “defined” a relation which lies in the middle of the greatest and least fixed point. To see why this argument is true, we found this theorem as an equation could be used as a possible definition of strong equivalence: (c.f. p. 49 of [1])

Definition 3. Define recursively a new behavioral relation $\sim' \in Q \times Q$ as follows: $q_1 \sim' q_2$ if and only if for all $\mu \in A$

- $\forall q'_1$ such that $q_1 \xrightarrow{\mu} q'_1, \exists q'_2$ such that $q_2 \xrightarrow{\mu} q'_2$ and $q'_1 \sim' q'_2$,
- $\forall q'_2$ such that $q_2 \xrightarrow{\mu} q'_2, \exists q'_1$ such that $q_1 \xrightarrow{\mu} q'_1$ and $q'_1 \sim' q'_2$.

This is exactly the same as above “cases” theorem if the theorem were used as a definition of strong equivalence. Robin Milner calls this theorem the “property (*)” of strong equivalence. (c.f. p.88 of [5]) But as Prof. Gorrieri’s book [1] already told with examples: “this does not identify a unique relation, as many different relations satisfy this recursive definition.”, and the fact that any mathematical (or logic) definitions must precisely specify the targeting object (unless the possible covered range itself is a targeting object).

But why the recursive definition failed to get a largest bisimulation (strong equivalence)? The textbooks didn’t give an clear answer, but in the view of theorem proving, now it’s quite clear: such a recursive definition can only restrict the target relation into the range of all fixed points, while it’s the co-induction theorem finally restrict the target relation to the greatest solution. Without any of them, the solution will not be unique (thus it’s not a valid definition).

Now we prove the old (STRONG_EQUIV, \sim) and new definition (STRONG_EQ) of strong equivalence are equivalent, i.e.

STR_EQUIV_TO_STR_EQ:

$$\vdash E \sim E' \iff \text{STRONG_EQ } E \ E'$$

The proof of above theorem is the result when combining the proof for each directions:

STR_EQ_IMP_STR_EQUIV:

$$\vdash \text{STRONG_EQ } E \ E' \Rightarrow E \sim E'$$

STR_EQUIV_IMP_STR_EQ:

$$\vdash E \sim E' \Rightarrow \text{STRONG_EQ } E \ E'$$

The direction from the co-inductively defined STRONG_EQ to traditionally defined STRONG_EQUIV is relatively easy, the proof only depends on the definition of STRONG_EQUIV and the fact that STRONG_EQ is also a STRONG_BISIM relation:

STR_EQ_IS_STR_BISIM:

$$\vdash \text{STRONG_BISIM STRONG_EQ}$$

which can be easily proved by comparing the definition of STRONG_BISIM and the “cases” theorem generated from the co-inductively defined STRONG_EQ. Thus the maximality of strong equivalence is not needed.

The proof of the other direction, instead, must use the co-induction theorem STRONG_EQ_coind and the “property (*)” of STRONG_EQUIV mentioned once previously:

PROPERTY_STAR:

$$\vdash E \sim E' \iff$$

$\forall u.$

$$(\forall E_1. E \text{ --}u\text{--} E_1 \Rightarrow \exists E_2. E' \text{ --}u\text{--} E_2 \wedge E_1 \sim E_2) \wedge$$

$$\forall E_2. E' \text{ --}u\text{--} E_2 \Rightarrow \exists E_1. E \text{ --}u\text{--} E_1 \wedge E_1 \sim E_2$$

The proof of above “property (*)” is similar with previous steps, but it’s not trivial. Here we omit the details (the reader can always check the proof scripts for all details). For just once we show how co-induction theorem is used to prove STR_EQUIV_IMP_STR_EQ and we replay this proof by HOL’s interactive proof manager⁷:

⁷ in the following quote text, the leading > is the prompt of HOL running in PolyML, the function **g** puts an initial goal into the proof manager, and the function **e** applies tacticals to current goal

```

> g '!E E'. STRONG_EQUIV E E' ==> STRONG_EQ E E';
val it =
  Proof manager status: 1 proof.
1. Incomplete goalstack:
  Initial goal:

    !E E'. E ~ E' ==> STRONG_EQ E E'

:
  proofs

```

Now we're going to apply the co-induction theorem:

```

⊢ (∀ a0 a1.
  STRONG_EQ' a0 a1 ⇒
  ∀ u.
    (∀ E1.
      a0 --u-> E1 ⇒ ∃ E2. a1 --u-> E2 ∧ STRONG_EQ' E1 E2) ∧
    ∀ E2.
      a1 --u-> E2 ⇒ ∃ E1. a0 --u-> E1 ∧ STRONG_EQ' E1 E2) ⇒
  ∀ a0 a1. STRONG_EQ' a0 a1 ⇒ STRONG_EQ a0 a1)

```

The tactical for applying such (co)induction theorems in HOL is to reduce the goal using a supplied implication, with higher-order matching, this tactical is called `HO_MATCH_MP_TAC`:

```

> e (HO_MATCH_MP_TAC STRONG_EQ_coind);
OK..
1 subgoal:
val it =

!E E'.
  E ~ E' ==>
  !u.
    (!E1. E --u-> E1 ==> ?E2. E' --u-> E2 /\ E1 ~ E2) /\
    !E2. E' --u-> E2 ==> ?E1. E --u-> E1 /\ E1 ~ E2

:
  ?.proof

```

The rest steps is to use the “property (*)” to rewrite the right side of the implication:

```

> e (PURE_ONCE_REWRITE_TAC [GSYM PROPERTY_STAR]);
OK..
1 subgoal:
val it =

!E E'. E ~ E' ==> E ~ E'

:
  ?.proof

```

Now things get very clear, a simple rewrite with boolean theorems will solve the goal easily:

```

> e (RW_TAC bool_ss []);
OK..

Goal proved.
|- !E E'. E ~ E' ==> E ~ E'

Goal proved.
|- !E E'.
  E ~ E' ==>
  !u.
    (!E1. E --u-> E1 ==> ?E2. E' --u-> E2 /\ E1 ~ E2) /\
    !E2. E' --u-> E2 ==> ?E1. E --u-> E1 /\ E1 ~ E2

```

```

val it =
  Initial goal proved.
|- !E E'. E ~ E' ==> STRONG_EQ E E':
  ?.proof

```

Combining all the step together, a single Standard ML function in the proof script can be written to finish the proof and store the theorem with a name:

```

val STR_EQUIV_IMP_STR_EQ = store_thm (
  "STR_EQUIV_IMP_STR_EQ",
  "'!E E'. STRONG_EQUIV E E' ==> STRONG_EQ E E'",
  HO_MATCH_MP_TAC STRONG_EQ_coind (* co-induction principle used here! *)
>> REPEAT GEN_TAC
>> PURE_ONCE_REWRITE_TAC [GSYM PROPERTY_STAR]
>> RW_TAC bool_ss []);

```

5 Weak transition and weak equivalence

The formalization of weak bisimulation, together with weak (and rooted) equivalence (also called “observation equivalence” and “observation congruence” in old books) is minimal in this project. In this part, the main purpose is to define the weak equivalence co-inductively *first* and then prove the traditional definition (like **STRONG_EQUIV**) as a theorem. We want to convince the reader that, by using HOL’s coinduction facility, it’s much easier to get the same set of theorems like those for strong equivalence. These works are not part of the old CCS formalization in HOL88, it belongs to the author.⁸

There’re multiple ways to define the concept of weak transitions used in the definition of weak bisimulation. In early approach like Milner’s book, the first step is to define a **EPS** relation, which indicates that between two processes there’s nothing but zero or more τ transitions. In HOL, this can be though a non-recursive inductive relation and a RTC operation:

```

⊢ EPS1 a0 a1 ⇔ a0 --τ-> a1
⊢ EPS = EPS1*

```

Once we have the **EPS** relation, the weak transition can be defined but a normal transition wrapped with two **EPS** transitions:

```

⊢ a0 ==a1=>> a2 ⇔
  ∃ E1 E2. EPS a0 E1 ∧ E1 --a1-> E2 ∧ EPS E2 a2

```

Modern textbooks like [1] directly uses “weak trace” transition for defining weak bisimulation, in which there’s only one action in the trace. Here are the definition of weak trace:

Definition 4. (*Weak trace*) For any LTS $TS = (Q, A \cup \tau, \rightarrow)$, where $\tau \notin A$, define relation $\Rightarrow \subset Q \times A^* \times Q$ as the weak reflexive and transitive closure of \rightarrow , i.e., as the least relation induced by the following axiom and rules, where ϵ is the empty trace:

$$\begin{array}{c}
\frac{q_1 \xrightarrow{\alpha} q_2}{q_1 \Rightarrow q_2} \qquad \frac{q_1 \xrightarrow{\tau} q_2}{q_1 \xRightarrow{\epsilon} q_2} \qquad \frac{}{q \xRightarrow{\epsilon} q} \qquad \frac{q_1 \xRightarrow{\sigma_1} q_2 \quad q_2 \xRightarrow{\sigma_2} q_3}{q_1 \xRightarrow{\sigma_1 \sigma_2} q_3}
\end{array}$$

⁸ Of course, we can also rewrite the proof scripts for strong equivalence and fully benefits from HOL’s coinductive relation facility, but this is not very useful. The central idea of theorem proving is, once a theorem is successfully proved, its statement can be saved into disk for later use without the need to run the proof again everytime when it’s used. And the proof steps are not saved at all. Thus, the same theorem proved by different methods, when they were saved into disk, there’s absolutely no difference except for their names. On the other side, we want to keep the old definition for at least strong equivalence, because it’s a literal formalization of the definitions in standard textbooks.

In HOL, we can use a list of `Label` to represent the trace (thus there's naturally no τ in the list, as τ is not part of the type `Label` but `Action`) and empty list can be seen as the ϵ . As the result, the relation `WEAK_TRACE` has type `CCS -> Label list -> CCS -> bool`. Below is the rules theorem generated by `Hol_reln` command:

$$\begin{aligned} &\vdash (\forall E. \text{WEAK_TRACE } E \in E) \wedge \\ &\quad (\forall E \ E'. E \text{ --}\tau\text{--> } E' \Rightarrow \text{WEAK_TRACE } E \in E') \wedge \\ &\quad (\forall E \ E' \ l. E \text{ --label } l\text{--> } E' \Rightarrow \text{WEAK_TRACE } E \ [l] \ E') \wedge \\ &\quad \forall E_1 \ E_2 \ E_3 \ l_1 \ l_2. \\ &\quad \text{WEAK_TRACE } E_1 \ l_1 \ E_2 \wedge \text{WEAK_TRACE } E_2 \ l_2 \ E_3 \Rightarrow \\ &\quad \text{WEAK_TRACE } E_1 \ (l_1 ++ l_2) \ E_3 \end{aligned}$$

Now we take a look at the definition of weak bisimulation:

Definition 5. (*Weak bisimulation and weak equivalence*) For any LTS $(Q, A \cup \tau, \rightarrow)$, where $\tau \notin A$, a weak bisimulation is a relation $R \subset Q \times Q$ such that both R and its inverse R^{-1} are weak simulations. More explicitly, a weak bisimulation is a relation R such that if $(q_1, q_2) \in R$ then for all $\alpha \in A$

- $\forall q'_1$ such that $q_1 \xrightarrow{\alpha} q'_1, \exists q'_2$ such that $q_2 \xRightarrow{\alpha} q'_2$ and $(q'_1, q'_2) \in R$,
- $\forall q'_1$ such that $q_1 \xrightarrow{\tau} q'_1, \exists q'_2$ such that $q_2 \xRightarrow{\tau} q'_2$ and $(q'_1, q'_2) \in R$,

and, symmetrically,

- $\forall q'_2$ such that $q_2 \xrightarrow{\alpha} q'_2, \exists q'_1$ such that $q_1 \xRightarrow{\alpha} q'_1$ and $(q'_1, q'_2) \in R$,
- $\forall q'_2$ such that $q_2 \xrightarrow{\tau} q'_2, \exists q'_1$ such that $q_1 \xRightarrow{\tau} q'_1$ and $(q'_1, q'_2) \in R$.

States q and q' are weakly bisimilar (or weak bisimulation equivalent), denoted with $q \approx q'$, if there exists a weak bisimulation R such that $(q, q') \in R$.

There's no big problem to use all weak traces in above definition, as long as we limit the number of labels in the trace to just one. The real difficulty happens when we try to further define the *rooted weak bisimilarity* on top of weak equivalence, in which an auxiliary relation $q \xRightarrow{\tau} q'$ must be defined as

$$q \xRightarrow{\tau} q' \text{ if and only if } \exists q_1, q_2, q \xRightarrow{\epsilon} q_1 \xrightarrow{\tau} q_2 \xRightarrow{\epsilon} q'. \quad (1)$$

And the definition of rooted weak bisimilarity (in Milner's book it's also called "observational congruence"), noticed that it's not recursive:

Definition 6. (*Rooted weak bisimilarity*) Given an LTS $(Q, A \cup \tau, \rightarrow)$, two states q_1 and q_2 are rooted weak bisimilar, denoted $q_1 \approx^c q_2$, if for all $\mu \text{ in } A \cup \{\tau\}$

- $\forall q'_1$ such that $q_1 \xrightarrow{\mu} q'_1, \exists q'_2$ such that $q_2 \xRightarrow{\mu} q'_2$ and $q'_1 \approx q'_2$,
- $\forall q'_2$ such that $q_2 \xrightarrow{\mu} q'_2, \exists q'_1$ such that $q_1 \xRightarrow{\mu} q'_1$ and $q'_1 \approx q'_2$.

But there's a type error here: τ is not a legal trace, thus a term like `WEAK_TRACE E [tau] E'` is illegal with the existing definition of `WEAK_TRACE`. If we enlarge the type of traces to `Action list`, then invalid traces like $\sigma_1 \tau \sigma_2$ will become legal in HOL terms. One dirty solution is to define another relation `WEAK_TRACE'` which takes single `Action` having above auxiliary relation as part of its definition. But it actually coincides with `WEAK_TRANS`.

Since the use of `WEAK_TRACE` will cause theorems from HOL's `listTheory` being used to handle the list of labels. To simplify things, in all these weak bisimulation variants, we only use `WEAK_TRANS` and `EPS`. Here is the definition of `WEAK_BISIM` (not recursive) in HOL:

$$\begin{aligned} &\vdash \text{WEAK_BISIM } Wbsm \iff \\ &\quad \forall E \ E'. \\ &\quad \quad Wbsm \ E \ E' \Rightarrow \\ &\quad \quad (\forall l. \\ &\quad \quad \quad (\forall E_1. \\ &\quad \quad \quad \quad E \text{ --label } l\text{--> } E_1 \Rightarrow \end{aligned}$$

$$\begin{aligned}
& \exists E_2. E' ==\text{label } l \Rightarrow E_2 \wedge Wbsm E_1 E_2) \wedge \\
& \forall E_2. \\
& \quad E' \text{ --label } l \rightarrow E_2 \Rightarrow \\
& \quad \exists E_1. E ==\text{label } l \Rightarrow E_1 \wedge Wbsm E_1 E_2) \wedge \\
& (\forall E_1. E \text{ --}\tau\text{--} \rightarrow E_1 \Rightarrow \exists E_2. EPS E' E_2 \wedge Wbsm E_1 E_2) \wedge \\
& \forall E_2. E' \text{ --}\tau\text{--} \rightarrow E_2 \Rightarrow \exists E_1. EPS E E_1 \wedge Wbsm E_1 E_2
\end{aligned}$$

And the relation WEAK_EQUIV is co-inductively defined by HOL's `Hol_coreln` command:

```

val (WEAK_EQUIV_rules, WEAK_EQUIV_coind, WEAK_EQUIV_cases) = Hol_coreln '
  (!E E'.
    (!l.
      (!E1. TRANS E (label l) E1 ==>
        (?E2. WEAK_TRANS E' (label l) E2 /\ WEAK_EQUIV E1 E2)) /\
      (!E2. TRANS E' (label l) E2 ==>
        (?E1. WEAK_TRANS E (label l) E1 /\ WEAK_EQUIV E1 E2))) /\
      (!E1. TRANS E tau E1 ==> (?E2. EPS E' E2 /\ WEAK_EQUIV E1 E2)) /\
      (!E2. TRANS E' tau E2 ==> (?E1. EPS E E1 /\ WEAK_EQUIV E1 E2))
    ==> WEAK_EQUIV E E')';

```

Like the strong equivalence, above command generates three theorems which fully characteristics the weak equivalence relation:

1. The (forward) rules for weak equivalence:

$$\begin{aligned}
& \vdash (\forall l. \\
& \quad (\forall E_1. \\
& \quad \quad E \text{ --label } l \rightarrow E_1 \Rightarrow \\
& \quad \quad \exists E_2. E' ==\text{label } l \Rightarrow E_2 \wedge E_1 \approx E_2) \wedge \\
& \quad \forall E_2. \\
& \quad \quad E' \text{ --label } l \rightarrow E_2 \Rightarrow \exists E_1. E ==\text{label } l \Rightarrow E_1 \wedge E_1 \approx E_2) \wedge \\
& (\forall E_1. E \text{ --}\tau\text{--} \rightarrow E_1 \Rightarrow \exists E_2. EPS E' E_2 \wedge E_1 \approx E_2) \wedge \\
& (\forall E_2. E' \text{ --}\tau\text{--} \rightarrow E_2 \Rightarrow \exists E_1. EPS E E_1 \wedge E_1 \approx E_2) \Rightarrow \\
& E \approx E'
\end{aligned}$$

2. The co-induction theorem which assert the maximality of the relation:

$$\begin{aligned}
& \vdash (\forall a_0 a_1. \\
& \quad WEAK_EQUIV' a_0 a_1 \Rightarrow \\
& \quad (\forall l. \\
& \quad \quad (\forall E_1. \\
& \quad \quad \quad a_0 \text{ --label } l \rightarrow E_1 \Rightarrow \\
& \quad \quad \quad \exists E_2. a_1 ==\text{label } l \Rightarrow E_2 \wedge WEAK_EQUIV' E_1 E_2) \wedge \\
& \quad \quad \forall E_2. \\
& \quad \quad \quad a_1 \text{ --label } l \rightarrow E_2 \Rightarrow \\
& \quad \quad \quad \exists E_1. a_0 ==\text{label } l \Rightarrow E_1 \wedge WEAK_EQUIV' E_1 E_2) \wedge \\
& \quad \quad (\forall E_1. \\
& \quad \quad \quad a_0 \text{ --}\tau\text{--} \rightarrow E_1 \Rightarrow \exists E_2. EPS a_1 E_2 \wedge WEAK_EQUIV' E_1 E_2) \wedge \\
& \quad \quad \forall E_2. a_1 \text{ --}\tau\text{--} \rightarrow E_2 \Rightarrow \exists E_1. EPS a_0 E_1 \wedge WEAK_EQUIV' E_1 E_2) \Rightarrow \\
& \quad \forall a_0 a_1. WEAK_EQUIV' a_0 a_1 \Rightarrow a_0 \approx a_1
\end{aligned}$$

3. The “cases” theorem (or “property (*)”) for weak equivalence:

$$\begin{aligned}
& \vdash a_0 \approx a_1 \iff \\
& (\forall l. \\
& \quad (\forall E_1. \\
& \quad \quad a_0 \text{ --label } l \rightarrow E_1 \Rightarrow \\
& \quad \quad \exists E_2. a_1 ==\text{label } l \Rightarrow E_2 \wedge E_1 \approx E_2) \wedge \\
& \quad \forall E_2. \\
& \quad \quad a_1 \text{ --label } l \rightarrow E_2 \Rightarrow \\
& \quad \quad \exists E_1. a_0 ==\text{label } l \Rightarrow E_1 \wedge E_1 \approx E_2) \wedge \\
& (\forall E_1. a_0 \text{ --}\tau\text{--} \rightarrow E_1 \Rightarrow \exists E_2. EPS a_1 E_2 \wedge E_1 \approx E_2) \wedge \\
& \forall E_2. a_1 \text{ --}\tau\text{--} \rightarrow E_2 \Rightarrow \exists E_1. EPS a_0 E_1 \wedge E_1 \approx E_2
\end{aligned}$$

Unlike in the definition of strong equivalence, our definition of `WEAK_EQUIV` is unrelated to the definition of `WEAK_BISIM`. But we want to show that, the textbook definition for weak equivalence which is similar with the definition of strong equivalence

$$\vdash E \sim E' \iff \exists Bsm. Bsm\ E\ E' \wedge \text{STRONG_BISIM}\ Bsm$$

can now be proved as a theorem:

$$\vdash E \approx E' \iff \exists Wbsm. Wbsm\ E\ E' \wedge \text{WEAK_BISIM}\ Wbsm$$

The proof is very simple, because now we have the “property (*)” for free. The first step is to prove that the weak equivalence is also a weak bisimulation relation:

$$\vdash \text{WEAK_BISIM}\ \text{WEAK_EQUIV}$$

Then in the proof of `WEAK_EQUIV`, one direction can be easily proved by above theorem, and other direction can also be proved easily by co-induction theorem and first-order proof searching (using HOL’s `METIS_TAC` [12]):

```
(* Alternative definition of WEAK_EQUIV, similar with STRONG_EQUIV (definition).
   "Weak bisimilarity contains all weak bisimulations (thus maximal)"
   *)
val WEAK_EQUIV = store_thm ("WEAK_EQUIV",
  ‘‘!E E’. WEAK_EQUIV E E’ = (?Wbsm. Wbsm E E’ /\ WEAK_BISIM Wbsm)‘‘,
  REPEAT GEN_TAC
>> EQ_TAC (* 2 sub-goals here *)
>| [ (* goal 1 (of 2) *)
    DISCH_TAC \
    EXISTS_TAC ‘‘WEAK_EQUIV‘‘ \
    ASM_REWRITE_TAC [WEAK_EQUIV_IS_WEAK_BISIM],
    (* goal 2 (of 2) *)
    Q.SPEC_TAC (‘E’, ‘E’) \
    Q.SPEC_TAC (‘E’, ‘E’) \
    HO_MATCH_MP_TAC WEAK_EQUIV_coind \
    METIS_TAC [WEAK_BISIM] ]);
```

As mentioned in the literal [10], bisimulation equivalence is one of the most well-studied co-inductive relation. But since HOL (and other theorem provers like Coq and Isabelle) started to support the co-inductive relation features (in very recent years), the correctness of these features were never confirmed on the (strong and weak) bisimulation equivalence defined on CCS-like graph structures. Now in this project, we have finally done this work.

Finally the rooted weak equivalence is defined in HOL as follows: (again, not recursive)

$$\begin{aligned} \vdash E \approx^c E' &\iff \\ \forall u. & \\ (\forall E_1. E \text{ --}u\text{--} E_1 \Rightarrow \exists E_2. E' ==u\text{--}>> E_2 \wedge E_1 \approx E_2) \wedge & \\ \forall E_2. E' \text{ --}u\text{--} E_2 \Rightarrow \exists E_1. E ==u\text{--}>> E_1 \wedge E_1 \approx E_2 & \end{aligned}$$

5.1 Laws for strong equivalence

Based on the definition of `STRONG_EQUIV` and SOS inference rules for the `TRANS` relation, we have proved a large set of theorems concerning the strong equivalence of CCS processes. Below is a list of fundamental congruence theorems for strong equivalence:

STRONG_EQUIV_SUBST_PREFIX:

$$\vdash E \sim E' \Rightarrow \forall u. u..E \sim u..E'$$

STRONG_EQUIV_PRESERVED_BY_SUM:

$$\vdash E_1 \sim E'_1 \wedge E_2 \sim E'_2 \Rightarrow E_1 + E_2 \sim E'_1 + E'_2$$

STRONG_EQUIV_PRESERVED_BY_PAR:

$$\vdash E_1 \sim E'_1 \wedge E_2 \sim E'_2 \Rightarrow E_1 \parallel E_2 \sim E'_1 \parallel E'_2$$

STRONG_EQUIV_SUBST_RESTRICT:

$$\vdash E \sim E' \Rightarrow \forall L. \nu L E \sim \nu L E'$$

STRONG_EQUIV_SUBST_RELAB:

$$\vdash E \sim E' \Rightarrow \forall rf. \text{relab } E \text{ } rf \sim \text{relab } E' \text{ } rf$$

STRONG_SUM_IDENT_R:

Noticed that, the strong bisimulation equivalence is co-inductively defined, and two processes are strong equivalent if there's a bisimulation containing them. Thus, to prove two processes are strong equivalent, it's enough to find a bisimulation containing them. To prove they're not strong equivalent, it's enough to try to construct a bisimulation starting from them and the proof is finished whenever the attempt fails. In any case, there's no need to do induction on the data type of involved CCS processes.

Here are the strong laws proved for the sum operator:

$$\begin{array}{ll} \text{STRONG_SUM_IDENT_R:} & \vdash E + \text{nil} \sim E \\ \text{STRONG_SUM_IDEMP:} & \vdash E + E \sim E \\ \text{STRONG_SUM_COMM:} & \vdash E + E' \sim E' + E \\ \text{STRONG_SUM_IDENT_L:} & \vdash \text{nil} + E \sim E \\ \text{STRONG_SUM_ASSOC_R:} & \vdash E + E' + E'' \sim E + (E' + E'') \\ \text{STRONG_SUM_ASSOC_L:} & \vdash E + (E' + E'') \sim E + E' + E'' \\ \text{STRONG_SUM_MID_IDEMP:} & \vdash E + E' + E \sim E' + E \\ \text{STRONG_LEFT_SUM_MID_IDEMP:} & \vdash E + E' + E'' + E' \sim E + E'' + E' \end{array}$$

The strong laws for the par operator:

$$\begin{array}{ll} \text{STRONG_PAR_IDENT_R:} & \vdash E \parallel \text{nil} \sim E \\ \text{STRONG_PAR_COMM:} & \vdash E \parallel E' \sim E' \parallel E \\ \text{STRONG_PAR_IDENT_L:} & \vdash \text{nil} \parallel E \sim E \\ \text{STRONG_PAR_ASSOC:} & \vdash E \parallel E' \parallel E'' \sim E \parallel (E' \parallel E'') \\ \text{STRONG_PAR_PREF_TAU:} & \vdash u..E \parallel \tau..E' \sim u..(E \parallel \tau..E') + \tau..(u..E \parallel E') \\ \text{STRONG_PAR_TAU_PREF:} & \vdash \tau..E \parallel u..E' \sim \tau..(E \parallel u..E') + u..(\tau..E \parallel E') \\ \text{STRONG_PAR_TAU_TAU:} & \vdash \tau..E \parallel \tau..E' \sim \tau..(E \parallel \tau..E') + \tau..(\tau..E \parallel E') \\ \text{STRONG_PAR_PREF_NO_SYNCR:} & \\ \vdash l \neq \text{COMPL } l' \Rightarrow & \\ \quad \forall E \ E'. & \\ \quad \text{label } l..E \parallel \text{label } l'..E' \sim & \\ \quad \text{label } l..(E \parallel \text{label } l'..E') + & \\ \quad \text{label } l'..(\text{label } l..E \parallel E') & \end{array}$$

STRONG_PAR_PREF_SYNCR:

$$\begin{array}{ll} \vdash (l = \text{COMPL } l') \Rightarrow & \\ \quad \forall E \ E'. & \\ \quad \text{label } l..E \parallel \text{label } l'..E' \sim & \\ \quad \text{label } l..(E \parallel \text{label } l'..E') + & \\ \quad \text{label } l'..(\text{label } l..E \parallel E') + \tau..(E \parallel E') & \end{array}$$

The strong laws for the restriction operator:

$$\begin{array}{ll} \text{STRONG_RESTR_NIL:} & \vdash \nu L \text{nil} \sim \text{nil} \\ \text{STRONG_RESTR_SUM:} & \vdash \nu L (E + E') \sim \nu L E + \nu L E' \\ \text{STRONG_RESTR_PREFIX_TAU:} & \vdash \nu L (\tau..E) \sim \tau..\nu L E \\ \text{STRONG_RESTR_PR_LAB_NIL:} & \vdash l \in L \vee \text{COMPL } l \in L \Rightarrow \forall E. \nu L (\text{label } l..E) \sim \text{nil} \\ \text{STRONG_RESTR_PREFIX_LABEL:} & \vdash l \notin L \wedge \text{COMPL } l \notin L \Rightarrow \\ & \forall E. \nu L (\text{label } l..E) \sim \text{label } l..\nu L E \end{array}$$

The strong laws for the relabeling operator:

$$\begin{array}{ll} \text{STRONG_RELAB_NIL:} & \vdash \text{relab nil } rf \sim \text{nil} \\ \text{STRONG_RELAB_SUM:} & \vdash \text{relab } (E + E') \text{ } rf \sim \text{relab } E \text{ } rf + \text{relab } E' \text{ } rf \\ \text{STRONG_RELAB_PREFIX:} & \vdash \text{relab } (u..E) \text{ } (\text{RELAB } labl) \sim \\ & \text{relabel } (\text{RELAB } labl) \text{ } u..\text{relab } E \text{ } (\text{RELAB } labl) \end{array}$$

The strong laws for the recursion operator (for constants):

STRONG_UNFOLDING: $\vdash \text{rec } X \ E \sim \text{CCS_Subst } E \ (\text{rec } X \ E) \ X$
 STRONG_PREF_REC_EQUIV: $\vdash u.. \text{rec } s \ (v..u.. \text{var } s) \sim \text{rec } s \ (u..v.. \text{var } s)$
 STRONG_REC_ACT2: $\vdash \text{rec } s \ (u..u.. \text{var } s) \sim \text{rec } s \ (u.. \text{var } s)$

5.2 Expansion Law for strong equivalence

The final big piece of proof work in this project, is the representation and proof of the following *expansion law* (sometimes also called the *interleaving law*):

Proposition 1. (*Expansion Law*) Let $p = \sum_{i=1}^n \mu_i.p_i$ and $q = \sum_{j=1}^m \mu'_j.q_j$. Then

$$p|q \sim \sum_{i=1}^n \mu_i.(p_i|q) + \sum_{j=1}^m \mu'_j.(p|q_j) + \sum_{i,j:\bar{\mu}_i=\bar{\mu}'_j} \tau.(p_i|q_j) \quad (2)$$

Some characteristics made the formal proof very special and different from all other theorems that we have proved so far. For of all, numbers (of type `num`) were involved for the first time, and now our CCS theory depends on elementary mathematical theories provided by HOL, namely the `prim_recTheory` and `arithmeticTheory`. Although arithmetic operations like $+$, $-$, \cdot , $/$ were not involved (yet), but we do need to compare on number values and use some related theorems.

Also CCS accessors were defined and used to access the internal structure of CCS processes, namely `PREF_ACT` for getting the initial action and `PREF_PROC` for getting the rest of process without the first action. Together there's predicate `Is_Prefix` for testing if a CCS is a prefixed process:

$\vdash \text{PREF_ACT } (u..E) = u$
 $\vdash \text{PREF_PROC } (u..E) = E$
 $\vdash \text{Is_Prefix } E \iff \exists u \ E'. \ E = u..E'$

They are needed because we're going to represent $\mu_i.p_i$ as the value of a function: $f(i)$ in which f has the type `num` \rightarrow `CCS`. And in this way, to get μ_i and p_i we have to use accessors: "`PREF_ACT (f i)`" and "`PREF_PROC (f i)`".

The next job is to represent a finite sum of CCS processes. This is done by the following recursive function `SIGMA`:

$\vdash (\forall f. \text{SIGMA } f \ 0 = f \ 0) \wedge$
 $\forall f \ n. \text{SIGMA } f \ (\text{SUC } n) = \text{SIGMA } f \ n + f \ (\text{SUC } n)$

Thus if there's a function f of type `num` \rightarrow `CCS`, we should be able to represent $\sum_{i=1}^n f(i)$ by HOL term "`SIGMA f n`".

Now if we took a deeper look at the last summation of the right side of the expansion law, i.e. $\sum_{i,j:\bar{\mu}_i=\bar{\mu}'_j} \tau.(p_i|q_j)$, we found that such a "sum" cannot be represented directly, because there're two index i, j and their possible value pairs used in the sum depends on the synchronization of corresponding actions from each p_i and q_j . What we actually need is a recursively defined function taking all the p_i and q_j and return the synchronized process in forms like $\sum \tau.(p_i|q_j)$.

But this is still too complicated, instead we first define functions to synchronize just one process with another group of processes. This work is achieved by the function `SYNC` of type `Action` \rightarrow `CCS` \rightarrow (`num` \rightarrow `CCS`) \rightarrow `num` \rightarrow `CCS`:

$\vdash (\forall u \ P \ f.$
 $\quad \text{SYNC } u \ P \ f \ 0 =$
 $\quad \text{if } (u = \tau) \vee (\text{PREF_ACT } (f \ 0) = \tau) \text{ then nil}$
 $\quad \text{else if LABEL } u = \text{COMPL } (\text{LABEL } (\text{PREF_ACT } (f \ 0))) \text{ then}$
 $\quad \quad \tau..(P \ || \ \text{PREF_PROC } (f \ 0))$
 $\quad \text{else nil}) \wedge$
 $\forall u \ P \ f \ n.$
 $\quad \text{SYNC } u \ P \ f \ (\text{SUC } n) =$

```

if ( $u = \tau$ )  $\vee$  ( $\text{PREF\_ACT } (f \text{ (SUC } n)) = \tau$ ) then
   $\text{SYNC } u \text{ } P \text{ } f \text{ } n$ 
else if
   $\text{LABEL } u = \text{COMPL } (\text{LABEL } (\text{PREF\_ACT } (f \text{ (SUC } n))))$ 
then
   $\tau..(P \parallel \text{PREF\_PROC } (f \text{ (SUC } n))) + \text{SYNC } u \text{ } P \text{ } f \text{ } n$ 
else  $\text{SYNC } u \text{ } P \text{ } f \text{ } n$ 

```

Then the synchronization of two group of processes can be further defined by another recursive function ALL_SYNC of type $(\text{num} \rightarrow \text{CCS}) \rightarrow \text{num} \rightarrow (\text{num} \rightarrow \text{CCS}) \rightarrow \text{num} \rightarrow \text{CCS}$:

```

 $\vdash (\forall f \text{ } f' \text{ } m.$ 
   $\text{ALL\_SYNC } f \text{ } 0 \text{ } f' \text{ } m =$ 
   $\text{SYNC } (\text{PREF\_ACT } (f \text{ } 0)) (\text{PREF\_PROC } (f \text{ } 0)) \text{ } f' \text{ } m) \wedge$ 
 $\forall f \text{ } n \text{ } f' \text{ } m.$ 
   $\text{ALL\_SYNC } f \text{ (SUC } n) \text{ } f' \text{ } m =$ 
   $\text{ALL\_SYNC } f \text{ } n \text{ } f' \text{ } m +$ 
   $\text{SYNC } (\text{PREF\_ACT } (f \text{ (SUC } n))) (\text{PREF\_PROC } (f \text{ (SUC } n))) \text{ } f' \text{ } m$ 

```

Some lemmas about SIGMA and the two synchronization functions were proved first:

```

SIGMA_TRANS_THM_EQ:
 $\vdash \text{SIGMA } f \text{ } n \text{ } \text{--}u\text{--} \rightarrow E \iff \exists k. k \leq n \wedge f \text{ } k \text{ } \text{--}u\text{--} \rightarrow E$ 
SYNC_TRANS_THM_EQ:
 $\vdash \text{SYNC } u \text{ } P \text{ } f \text{ } m \text{ } \text{--}v\text{--} \rightarrow Q \iff$ 
 $\exists j \text{ } l.$ 
 $j \leq m \wedge (u = \text{label } l) \wedge$ 
 $(\text{PREF\_ACT } (f \text{ } j) = \text{label } (\text{COMPL } l)) \wedge (v = \tau) \wedge$ 
 $(Q = P \parallel \text{PREF\_PROC } (f \text{ } j))$ 
ALL_SYNC_TRANS_THM_EQ:
 $\vdash \text{ALL\_SYNC } f \text{ } n \text{ } f' \text{ } m \text{ } \text{--}u\text{--} \rightarrow E \iff$ 
 $\exists k \text{ } k' \text{ } l.$ 
 $k \leq n \wedge k' \leq m \wedge (\text{PREF\_ACT } (f \text{ } k) = \text{label } l) \wedge$ 
 $(\text{PREF\_ACT } (f' \text{ } k') = \text{label } (\text{COMPL } l)) \wedge (u = \tau) \wedge$ 
 $(E = \text{PREF\_PROC } (f \text{ } k) \parallel \text{PREF\_PROC } (f' \text{ } k'))$ 

```

Finally, we have proved the Expansion Law in the following form:

```

STRONG_PAR_LAW:
 $\vdash (\forall i. i \leq n \Rightarrow \text{Is\_Prefix } (f \text{ } i)) \wedge$ 
 $(\forall j. j \leq m \Rightarrow \text{Is\_Prefix } (f' \text{ } j)) \Rightarrow$ 
 $\text{SIGMA } f \text{ } n \parallel \text{SIGMA } f' \text{ } m \sim$ 
 $\text{SIGMA } (\lambda i. \text{PREF\_ACT } (f \text{ } i)..(\text{PREF\_PROC } (f \text{ } i) \parallel \text{SIGMA } f' \text{ } m))$ 
 $n +$ 
 $\text{SIGMA } (\lambda j. \text{PREF\_ACT } (f' \text{ } j)..(\text{SIGMA } f \text{ } n \parallel \text{PREF\_PROC } (f' \text{ } j)))$ 
 $m + \text{ALL\_SYNC } f \text{ } n \text{ } f' \text{ } m$ 

```

6 Missing pieces and Future directions

The old proof scripts provided by Prof. Nesi do not contain anything related to weak bisimulation, while these things were talked as major work in the original paper. We think those proof scripts must have been unfortunately lost. In our project, due to time limits we only re-defined the concepts of weak transitions, weak bisimulation, weak bisimulation equivalence (observation equivalence) and rooted weak equivalence (observation congruence), but almost didn't prove any useful results, except for the experiments to show the correctness of HOL's co-inductive relation defining facility (`Hol_coreln`) for weak bisimulation equivalences. Given the fact that, more practical model checking were done by comparing (rooted) weak bisimulation equivalences between two CCS processes, our current work is far from complete.

The other big missing piece is the decision procedure for automatic checking of strong (and weak) bisimulation equivalence. There exists some fast algorithms for bisimulation equivalence checking, they were mostly based on reductions of the equivalence checking to the so-called “coarest relational partitioning” problem. (c.f. [13] and [14]), some variants (e.g. [15]) are suitable for compact representations like CCS, in which the whole graph is not visible. On the other side, we know Concurrency Workbench didn’t use the most efficient algorithm (c.f. p.13 of [16]), and this leaves us a room to create a faster equivalence checking tool, and it runs even inside a theorem prover!

Thus, the author hopes to continue this project with the following possible direction:

1. Complete the theory for (rooted) weak equivalence and prove the related weak laws.
2. Create decision procedures for bisimulation equivalence checking which take two CCS processes and give a theorem about their equivalence.
3. Formally prove some deep theorems for bisimulation equivalences, e.g. the Hennessy Lemma (c.f. p.176 of [1])

Finally, any tool is only useful when it’s proven to be useful for resolving practical problems. But so far we haven’t shown anything for its applications. Indeed, maybe we can never show more useful results than those already have with software like Concurrency Workbench. So our main hope here, is to provide experiences and good basis for building more complicated process algebras. And when future researchers published new theorems in this area, maybe they could provide also formal proofs using the framework provided in this project.

7 Conclusions

In this exam paper (and project report), we have successfully ported the old formalization of process algebra CCS (no value passing, with explicit relabeling operator) from HOL88 to latest HOL4 (Kananaskis-11 and later). We started from the definition and syntax of CCS processes defined as inductive datatypes in Higher Order Logic, and then defined all the SOS (Structured Operational Semantics) inference rules as an relation `TRANS`. Then all the algebraic laws including the Expansion Theorem were proved on top of CCS datatype and SOS rules.

The other big work in this project is a single ML function (part of the old work, but we have fixed and enhanced the code) which could automatically compute the possible transitions for a given CCS process. Different with similar facility in softwares like Concurrency Workbench, the output of our function is a theorem. This is kind of trusting computing, as the only to build theorems is to construct it from other theorems. Although the correctness of such a program is not formally verified, but as long as it terminates with a theorem as output, the output **MUST** be correct. We have future plans to create a similar tool for equivalence checking.

This work is based on old CCS formalization in HOL88, done by Prof. Monica Nesi (of University of L’Aquila, Italy) in 1992-1995 when he was studying at University of Cambridge. Thanks to Prof. Nesi for finding and sending the old HOL88 proof scripts to the author.

Thanks to Prof. Roberto Gorrieri, who taught CCS and LTS theory to the author, and his supports on continuing this HOL-CCS project as exam project of his course.

Thanks to Prof. Andrea Asperti, who taught the interactive theorem proving techniques to the author, although it’s in another different theorem prover (Matita).

The paper is written in \LaTeX and LNCS template, with theorems generated automatically by HOL’s \TeX exporting module (`EmitTex`) from the proof scripts.

References

1. Gorrieri, R.: Introduction to Concurrency Theory: Transition Systems and CCS. Springer (2015)
2. Keller, R.M.: Formal verification of parallel programs. Communications of the Association for Computing Machinery **19**(7) (1976) 371–384
3. Milner, R.: A Calculus of Communicating Systems. Volume 92 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg (1980)
4. Gordon, M.: From LCF to HOL: a short history. Proof, language, and interaction (July 2000)

5. Milner, R.: Communication and concurrency (1989)
6. Nesi, M.: A formalization of the process algebra ccs in high order logic. Technical report, University of Cambridge, Computer Laboratory (1992)
7. Kreitz, C., Rahli, V.: Introduction to Classic ML. (August 2011) 1–18
8. Moller, F.: The Edinburgh Concurrency Workbench user manual (Version 7.1). (April 2017)
9. Melham, T.F.: A Package For Inductive Relation Definitions In HOL. In: 1991., International Workshop on the HOL Theorem Proving System and Its Applications, IEEE (1991) 350–357
10. Sangiorgi, D.: Introduction to Bisimulation and Coinduction. Cambridge University Press (October 2011)
11. : The HOL System DESCRIPTION. (March 2017) 1–349
12. Hurd, J.: First-order proof tactics in higher-order logic theorem provers. Design and Application of Strategies/Tactics in Higher ... (2003)
13. Kanellakis, P.C., Smolka, S.A.: CCS expressions, finite state processes, and three problems of equivalence. Information and Computation **86**(1) (1990) 43–68
14. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM Journal on Computing **16**(6) (1987) 973–989
15. Lee, D., Yannakakis, M.: Online minimization of transition systems (extended abstract). ACM, New York, New York, USA (July 1992)
16. CLEAVELAND, R., PARROW, J., STEFFEN, B.: The Concurrency Workbench - a Semantics-Based Tool for the Verification of Concurrent Systems. AcM Transactions on Programming Languages and Systems **15**(1) (January 1993) 36–72

Appendix A: running the proof scripts in HOL4

Suppose HOL (Kananaskis-11) has been installed⁹ and the two entry commands `hol` and `Holmake` have been made available in current Shell environment (e.g. their containing directory is in `PATH` environment variable). Copying all above 4 files into a empty directory and execute `Holmake`, they should be compiled correctly with some extra files generated.

Then after executing `hol`, enter the following commands to load all CCS related scripts:

```
-----
      HOL-4 [Kananaskis 11 (logknl, built Sat Apr 29 12:55:33 2017)]

      For introductory HOL help, type: help "hol";
      To exit type <Control>-D
-----
> load "ExampleTheory";
val it = (): unit
> open ExampleTheory;
...
>
```

Then we can either access already proved theorems storing into ML variables:

```
> STRONG_PAR_LAW;
val it =
  |- !f n f' m.
    (!i. i <= n ==> Is_Prefix (f i)) /\ (!j. j <= m ==> Is_Prefix (f' j)) ==>
    SIGMA f n || SIGMA f' m
~
    SIGMA (\i. PREF_ACT (f i)..(PREF_PROC (f i) || SIGMA f' m)) n +
    SIGMA (\j. PREF_ACT (f' j)..(SIGMA f n || PREF_PROC (f' j))) m +
    ALL_SYNC f n f' m:
thm
```

or compute CCS transitions from any given process:

⁹ For installation instructions of HOL4, see <https://hol-theorem-prover.org/#get>. To run the scripts mentioned in this paper correctly, please use kananaskis-11 (latest released version).


```

> CCS_TRANS ' '(nu "a") (In "a"..nil || Out "a"..nil)';
val it =
  (|- !u E.
    'm "a" (In "a"..nil || Out "a"..nil) --u-> E <=>
    (u = tau) /\ (E = 'm "a" (nil || nil)),
    [(('tau',
      'm "a" (nil || nil)')]):
    thm * (term * term) list

> CCS_TRANS ' '(In "a"..nil || Out "a"..nil)';
val it =
  (|- !u E.
    In "a"..nil || Out "a"..nil --u-> E <=>
    ((u = In "a") /\ (E = nil || Out "a"..nil) /\
      (u = Out "a") /\ (E = In "a"..nil || nil)) /\
    (u = tau) /\ (E = nil || nil),
    [(('In "a"',
      'nil || Out "a"..nil'),
      ('Out "a"',
      'In "a"..nil || nil'),
      ('tau',
      'nil || nil'))]:
    thm * (term * term) list

```

The generated files *.sig contain lists of all proved theorems, the HTML versions were also generated as *.html files.