λ_{lock} encoding to π-calculus

S. Wortelboer
Supervisors: J.A. Pérez, D. Frumin

June 2024

Abstract

λ_{lock}, by Jacobs et al [3], is a deadlock-free functional language that introduces concurrent processing with resource sharing through locks. CLASS, by Rocha [4], achieves the same through an extension of a linear π-calculus called μCLL. This project develops an encoding from λ_{lock} to CLASS. The necessity comes from the fact that λ_{lock} is not confluent and has a degree of non-determinism that makes it hard to verify the outcomes of any program. CLASS also captures the non-deterministic behaviour of locks (called cells) but does not make a choice and hence preserves the property of confluence. Any reduction in CLASS thus captures all possible outcomes. To allow the development of the encoding CLASS is extended into CLASS+, which captures the ownership property of λ_{lock} and the behaviour of discarding locks while preserving the stored values.

1 Introduction

λ_{lock} [3] is a function language that adds concurrent processing with resource sharing through locks. It adds non-deterministic behaviour to an otherwise deterministic calculus. The non-determinism comes as the semantics are defined in a non-confluent way. In the case that multiple threads are competing to acquire a lock, for example, the choice of which reduction to apply first can affect the outcome with the alternative being lost. Even in the case that all possible reductions are taken into account, the way reductions are applied it is not entirely obvious what the next step will be. For example, a lock interaction for a thread might be hidden behind a pure reduction, but it will still be competing, moreover, it leads to different paths of reductions that also lead to the same result. This is counter intuitive, hence there is a need to have the property of confluence.

CLASS [4] is a linear π-calculus that extends μCLL, which is relates via propositions-as-type [1, 2, 5] to second-order classical linear logic, with affine resources and cells. Cells are very similar to λ_{lock} locks. Both can store and retrieve values, can be shared among multiple threads/processes with the same constraints, and both use session types to ensure deadlock freedom in the same manner between empty and full states. CLASS also has the property of confluence and is the perfect candidate for a translation target. CLASS does have additional affine channels, channels which can be used or discarded. These form the foundation for channels stored in cells. As a consequence, CLASS does not have any concept of ownership and simply dismisses the channel once all processes have released their reference. This issue needs to be overcome in order to facilitate the encoding.

In this project, first CLASS+ will be introduced. CLASS+ will add the concept of ownership to cells while keeping affine channels. The modified types, type rules and operational semantics are then given which preserve the properties of type preservation, progress and confluence as each of these properties is an important prerequisite for the encoding. Type preservation is a fundamental property to have for any language, while both languages have deadlock freedom which is synonymous with progress, and finally, confluence is the property that is the main purpose of this translation. Once CLASS is extended, the construction of the encoding is given by the definition of the encoding of types, type rules and processes. The encoding is given as a proof of translation of the type rules which proofs type preservation. Furthermore, the encoding for processes is given to enable the validation of preservation of semantics (i.e. soundness and completeness). The validation of the latter is assumed, examples to illustrate that such properties hold are given.

1.1 λ_{lock}

λ_{lock} uses for its foundation a typed λ-calculus that differentiates between linear and unrestricted variables and functions. It then extends this calculus by adding currency and shared resources through locks, which retain deadlock freedom through the principle definition that locks connect/reference to locks and create an acyclic graph. Furthermore, each lock has one owning reference and zero or more child references.

To facilitate concurrency and shared resources, it is necessary to deviate from the traditional way of writing the operational semantics. A set of threads executing expressions combined with locks are introduced. On such a set a reduction can take place which affects one or more items in the set. For the pure expressions inherited from the base calculus, one thread reduces its expression, while the expressions related to locks always act or generate multiple items in the set. Such a set will be called a process and its structure always adheres to the defined principle of λ_{lock} as stated in the previous paragraph. The operational semantics follow the form given below.

$$\left\{\begin{array}{c} t_1 \mapsto \text{Thread}(e_1) \\ \ldots \\ t_n \mapsto \text{Thread}(e_n) \\ k_1 \mapsto \text{Lock}(c_1, v_1) \\ \ldots \\ k_m \mapsto \text{Lock}(c_m, v_m) \end{array}\right\} \overset{i}{\rightsquigarrow} \left\{\begin{array}{c} t_1 \mapsto \text{Thread}(e_1) \\ \ldots \\ t_{n'} \mapsto \text{Thread}(e_{n'}) \\ k_1 \mapsto \text{Lock}(c_1, v_1) \\ \ldots \\ k_{m'} \mapsto \text{Lock}(c_{m'}, v_{m'}) \end{array}\right\}$$

While λ-calculus is confluent, meaning the order of reduction does not affect the outcome, λ_{lock} adds non-deterministic behaviour that voids this property. Due to competition between threads depending on the order of reduction, different results may be achieved. This can be demonstrated using a small example below. In this example, there are only two possibilities for reduction, acquire can applied to t_1 and k or t_2 and k. Depending on which reduction gets priority the order of the multiplication or addition creates a different outcome for the eventual value stored in the lock.

$$\left\{\begin{array}{c} t_1 \mapsto \text{Thread}(K_1[\textbf{let } x, y = \textbf{acquire}(\langle k \rangle) \textbf{ in } \textbf{release}(x, 2 * y)]) \\ t_2 \mapsto \text{Thread}(K_2[\textbf{let } x, y = \textbf{acquire}(\langle k \rangle) \textbf{ in } \textbf{release}(x, 2 + y)]) \\ k \mapsto \text{Lock}(1, \text{Some}(1)) \end{array}\right\} \rightsquigarrow^*$$

$$\left\{\begin{array}{c} t_1 \mapsto \text{Thread}(K_1[\textbf{release}(x, 2 * 1)]) \\ t_2 \mapsto \text{Thread}(K_2[\textbf{let } x, y = \textbf{acquire}(\langle k \rangle) \textbf{ in } \textbf{release}(x, 2 + y)]) \\ k \mapsto \text{Lock}(1, \text{None}) \end{array}\right\} \textit{or}$$

$$\left\{\begin{array}{c} t_1 \mapsto \text{Thread}(K_1[\textbf{let } x, y = \textbf{acquire}(\langle k \rangle) \textbf{ in } \textbf{release}(x, 2 * y)]) \\ t_2 \mapsto \text{Thread}(K_2[\textbf{release}(x, 2 + 1)]) \\ k \mapsto \text{Lock}(1, \text{None}) \end{array}\right\} \rightsquigarrow^*$$

$$\left\{\begin{array}{c} \ldots \\ k \mapsto \text{Lock}(1, \text{Some}(4))) \end{array}\right\} \textit{or}$$

$$\left\{\begin{array}{c} \ldots \\ k \mapsto \text{Lock}(1, \text{Some}(6))) \end{array}\right\}$$

The principle that at most two threads can have at most one lock in common also is evident in the reductions. Any thread can create new locks at any time, but when it wants to share a lock with some other thread it can only be done by spawning a new thread by forking on the lock. To ensure correctness session types will ensure that the expressions always follow the proper protocol and it can not achieve a state where a lock remains empty with no thread to release it. Furthermore, while forking will expand the graph, it remains possible to change the topology by sharing locks through locks. The property of linearity is used to ensure that unsafe behaviour that violates the principle is impossible. This can be demonstrated in the example below. Here the reference of lock k_2 from thread t_1 that also is referenced by t_3 is stored into lock k_1 also referenced by thread t_2, which later acquires the lock k_2. This alters the topology by removing the edge from t_1 to k_2 and adds t_2 to k_2.

$$\left\{\begin{array}{c} t_1 \mapsto \mathrm{Thread}(K_1[\mathbf{release}(\langle k_1\rangle, \langle k_2\rangle)]) \\ t_2 \mapsto \mathrm{Thread}(K_2[\mathbf{acquire}(\langle k_1\rangle)]) \\ t_3 \mapsto \mathrm{Thread}(K_3[\langle k_2\rangle]) \\ k_1 \mapsto \mathrm{Lock}(c_1, \mathrm{None}) \\ k_2 \mapsto \mathrm{Lock}(c_2, v) \end{array}\right\} \overset{k_1}{\rightsquigarrow}$$

$$\left\{\begin{array}{c} t_1 \mapsto \mathrm{Thread}(K_1[\langle k_1\rangle]) \\ t_2 \mapsto \mathrm{Thread}(K_2[\mathbf{acquire}(\langle k_1\rangle)]) \\ t_3 \mapsto \mathrm{Thread}(K_3[\langle k_2\rangle]) \\ k_1 \mapsto \mathrm{Lock}(c_1, \mathrm{Some}(\langle k_2\rangle)) \\ k_2 \mapsto \mathrm{Lock}(c_2, v) \end{array}\right\} \overset{k_1}{\rightsquigarrow}$$

$$\left\{\begin{array}{c} t_1 \mapsto \mathrm{Thread}(K_1[\langle k_1\rangle]) \\ t_2 \mapsto \mathrm{Thread}(K_2[(\langle k_1\rangle, \langle k_2\rangle)]) \\ t_3 \mapsto \mathrm{Thread}(K_3[\langle k_2\rangle]) \\ k_1 \mapsto \mathrm{Lock}(c_1, \mathrm{None}) \\ k_2 \mapsto \mathrm{Lock}(c_2, v) \end{array}\right\}$$

2 CLASS+

Cells in CLASS are very similar in their semantics to that of locks in λ_{lock}. As a consequence, most parts can be translated quite literally. However, where they differ is in how they treat their resources at the end of their lifecycle. λ_{lock} uses the property of ownership to return a value stored in a lock before it is destroyed. CLASS on the other hand uses affine sessions and discards any value held inside a cell once all references are released. To facilitate the encoding this needs to be overcome.

The first thing one might think of is to use multiple cells to encode locks. CLASS only allows one cell usage to be shared among some processes so in order to encode something like fork it becomes necessary to wrap the usages inside a new cell. As a consequence, a competition between procedures related to the reference counter and resource held will manifest itself in the encoded processes. These two values of a lock are updated independently, so having a process that deals with these values sequentially would not make much sense. Moreover, capturing such behaviour would cause the reduction in CLASS to branch out into a larger sum of processes, some of which relate to each other and produce a duplicate outcome.

One alternative is to develop a new lock process, however as cells already capture most of the desired behaviour it is easier to modify CLASS by adding the same behaviour through the property of ownership, making it possible to directly translate locks into cells. This section will introduce CLASS+ by extending the type system of CLASS with the addition of an owner parameter and a new action return. The operational semantics are extended, and the type rules are expanded. Finally, the most important properties related to the encoding are ensured by extending the proofs of type preservation, progress and confluence. It is also shown that CLASS+ is backwards compatible with CLASS by showing an encoding from CLASS to CLASS+ and proving preservation reduction through soundness and completeness.

2.1 Process Calculus and Operational Semantics

To be able to return the contents of a cell to a process insurance is needed that there is exactly one process that is designated to do so. To achieve this the type system is altered by the addition of a relation parameter for the usage and state types. The relation parameter lets a process fall in the category of owner or child, where the former is the designated process that receives the final contents of the cell. The grammar for cell types is replaced as follows (page 52, def. 10):

$$\begin{array}{lll} A, B ::= & \ldots & \\ & \mid S_f^o A \textit{ (owned full state)} & \mid U_f^o A \textit{ (owned full state)} \\ & \mid S_f^c A \textit{ (child full state)} & \mid U_f^c A \textit{ (child full state)} \\ & \mid S_e^o A \textit{ (owned empty state)} & \mid U_e^o A \textit{ (owned empty state)} \\ & \mid S_e^c A \textit{ (child empty state)} & \mid U_e^c A \textit{ (child empty state)} \end{array}$$

Furthermore, the definition of duality then becomes (page 53, def. 11):

$$\overline{S_f^o A} \triangleq U_f^o \overline{A}$$
$$\overline{S_f^c A} \triangleq U_f^c \overline{A}$$
$$\overline{S_e^o A} \triangleq U_e^o \overline{A}$$
$$\overline{S_e^c A} \triangleq U_e^c \overline{A}$$

The syntax of processing terms is extended with the definition of a new return operation, that captures the new feature (page 53, def. 12/fig. 3.1):

$$\begin{array}{rll} \mathcal{A}, \mathcal{B} ::= & \ldots & \\ & | \quad \textbf{return } c(a); P & \textit{(return)} \end{array}$$

The return operation allows the process to take the contents of a cell and drop the reference similar to release such that the cell can be destroyed, but only if every other process with a reference to the cell has released it first.

2.1.1 Operational Semantics

The operational semantics is made up of structural congruence and reduction rules, as per definitions 13 and 14 respectively. Definition 13 is extended by the addition of the following rule:

$$\textbf{share } x\{\textbf{return } x(a); P || \textbf{take } x(a); Q\} \equiv \textbf{take } x(a); \textbf{share } x\{\textbf{return } x(a); P || Q\} \quad [RTSh]$$

The rule allows for the process using the cell as a child to continue using the cell, while the process using the cell as an owner to wait for these processes to finish up. The choice for congruence as opposed to reduction, specifically for the case right-to-left, can be justified in multiple ways. First, a reduction does not make sense as nothing has happened, the process has not progressed, this only happens when the interaction between a full cell and the take takes place through the reduction $[S_f U_f t]$. In addition, the rule is similar to the rules $[TSh]$ and $[PSh]$, in particular the latter. It prioritizes put by taking it outside the share, which is essentially the same as with $[RTSh]$.

Definition 14(structural reductions) is modified by replacing $[S_f U_f f]$ with:

$$\textbf{cut } \{\textbf{cell } c(a.P) | c | \textbf{return } c(a'); Q\} \rightarrow \textbf{cut}\{P | a | \{a/a'\} Q\} \quad [S_f U_f f]$$

It proposes when all child processes have released their references using the structural congruence rule $[RSh]$, an owner process can then both fetch the value and discard the cell which was not possible before. For the owner, the option to discard the value still remains, and it is useful to add an alias for this behaviour with respect to the reduction $[\wedge \vee d]$:

$$\textbf{release } x \triangleq \textbf{return } x(a); \textbf{discard } a$$

2.1.2 Type System

The type system is extended by adding the relation type to the existing rules and the introduction of a new type rule for the new operation. For the relation type on states and usages of cells, there will be 3 possible combinations, which would expand the existing 3 share rules into 9 new rules. Instead, a combination operation that is both associative and commutative is defined for the sub-types, which would allow the definition of one general share rule.

$$\begin{array}{cc} S_{s_1}^{r_1} \quad U_{s_2}^{r_2} & r_1, r_2 \in \mathcal{R} \quad s_1, s_2 \in \mathcal{S} \\ \mathcal{S} = \{e, f\} & \mathcal{R} = \{o, c\} \\ e + f = e & o + c = o \\ f + e = e & c + o = o \\ f + f = f & c + c = c \\ e + e \neq & o + o \neq \end{array}$$

The new type semantics of definition 15/figure 3.4 then becomes:

$$\frac{P \vdash_\eta \vec{c} : U_f^{o/c}\vec{B}, \vec{a} : \vee\vec{C}, a : A; \Gamma}{\textbf{affine}_{\vec{c},\vec{a}}\, v; P \vdash_\eta \vec{c} : U_f^{o/c}\vec{B}, \vec{a} : \vee\vec{C}, a : \wedge A; \Gamma}\ Taffine$$

$$\frac{}{\textbf{discard}\ a \vdash_\eta a : \vee A; \Gamma}\ Tdiscard \quad \frac{Q \vdash_\eta \Delta a : A; \Gamma}{\textbf{use}\ a; Q \vdash_\eta \Delta, a : \vee A; \Gamma}\ Tuse$$

$$\frac{P \vdash_\eta \Delta, a : \wedge A; \Gamma}{\textbf{cell}\ c(a.P) \vdash_\eta \Delta, c : S_f^o A; \Gamma}\ Tcell \quad \frac{}{\textbf{empty}\ c \vdash_\eta c : S_e^o A; \Gamma}\ Tempty$$

$$\frac{}{\textbf{release}\ c \vdash_\eta c : U_f^c A; \Gamma}\ Trelease \quad \frac{Q \vdash_\eta \Delta, a : \vee A; \Gamma}{\textbf{return}\ c(a); Q \vdash_\eta \Delta, c : U_f^o A; \Gamma}\ Treturn$$

$$\frac{Q \vdash_\eta \Delta, a : \vee A, c : U_e^r A; \Gamma \quad r \in \mathcal{R}}{\textbf{take}\ c(a); Q \vdash_\eta \Delta, c : U_f^r A; \Gamma}\ Ttake \quad \frac{Q_1 \vdash_\eta \Delta_q, a : \wedge\overline{A}; \Gamma \quad Q_2 \vdash_\eta \Delta_2, c : U_f^r A; \Gamma \quad r \in \mathcal{R}}{\textbf{put}\ c(a.Q_1); Q_2 \vdash_\eta \Delta_1, \Delta_2, c : U_e^r A; \Gamma}\ Tput$$

$$\frac{P \vdash_\eta \Delta; \Gamma \quad Q \vdash_\eta \Delta; \Gamma}{P + Q \vdash_\eta \Delta; \Gamma}\ Tsum$$

$$\frac{P \vdash_\eta \Delta', c : U_{s_1}^{r_1} A; \Gamma \quad Q \vdash_\eta \Delta', c : U_{s_2}^{r_2} A; \Gamma}{\textbf{share}\ c\{P||Q\} \vdash_\eta \Delta', \Delta, c : U_{s_1+s_2}^{r_1+r_2} A; \Gamma}\ Tsh$$

The new share rule summarizes in one rule what would otherwise be 9 rules, with the basic takeaway being that at most one usage can be empty in the context of the content of the cell and at most one usage can be the owner in the context of relation to a cell irrespective what the context of the cell is and vice verse. The new definition also works well with the n-ary share construct and the commutative/linear distributive conversion of share as the combination operation defined for the content and relation parameters are commutative and associative.

The new rule [Treturn] also forces the rule [Trelease] to assume the usage relation of a child. This is necessary as otherwise it will break type preservation for the rule [RSh], as if we assume that the owner releases its reference it will have to promote the child to the position of owner. For example, a share where the owner releases and the child simply forwards, both the usage and state channel in the forward are of the type child. However, [RSh] tells us that the usage part becomes now the owner which would violate the definition of [Tfwd].

Finally, it has to be noted that it is still necessary for the contents of cells to be affine, otherwise, cells cannot be used by affine channels which would hamper the expressibility of CLASS+ and break backwards compatibility. It also would not make sense as any non-affine channel can be expressed as an affine channel, while it is not possible to do the same the other way around.

2.1.3 Type Preservation

The first thing that needs to be checked is the property of type preservation. Here only the new rules $[RTSh]$ and $[S_f U_f f]$ need to be proven such that the modifications satisfy theorem 1, where the following properties must hold:

1. If $P \vdash \Delta; \Gamma$ and $P \equiv Q$, then $Q \vdash \Delta; \Gamma$.
2. If $P \vdash \Delta; \Gamma$ and $P \to Q$, then $Q \vdash \Delta; \Gamma$.

Below is the proof extension for theorem 1(1). Only the newly added rules need to be evaluated, and the rules affected by the newly added relation parameter.

Case: $[Sh]$

The proof remains largely the same. The original proof for the usages shows the commutative property of the state parameter which is better captured in the new $[TSh]$, the new relation parameter has the same property and is independent from the state parameter. Which concludes the proof.

Note: the same evaluation can be applied to $[CSh]$, $[ShC!]$, $[ShM]$, $[ShSh]$, $[TSh]$ and $[PSh]$.

Case: $[RSh]$

The proof needs to be modified by adding the relation parameter. As the release (the action, not the alias) only can be used only by child processes the usage channel type has to reflect this. With this restriction, it leaves open all possibilities for the role of process P as it can still be an owner or child (i.e. $c+o$ or $c+c$) similar

to the state parameter. Hence the relation parameter in P simply assumes the value of that of the relation type of the share. Which concludes the proof.

Case: $[RTsh]$ left-to-right
$\mathbf{share}\ x\{\mathbf{return}\ x(a); P||\mathbf{take}\ x(a); Q\} \equiv \mathbf{take}\ x(a); \mathbf{share}\ x\{\mathbf{return}\ x(a); P||Q\}$

(1) $\Delta = \Delta_1, \Delta_2, x : U_f^o A$
(2) $\mathbf{return}\ x(a); P \vdash_\eta \Delta_1, x : U_f^o A; \Gamma$
(3) $\mathbf{take}\ x(a); Q \vdash_\eta \Delta_2, x : U_f^c A; \Gamma$, for some A, Δ_1, Δ_2
($[Tsh^{-1}]$ and left side)
(4) $Q \vdash_\eta \Delta_2, a : \vee A, x : U_e^c A; \Gamma$ ($[Ttake^{-1}]$ and (3))
(5) $\mathbf{share}\ x\{\mathbf{return}\ x(a); P||Q\} \vdash_\eta \Delta_1, \Delta_2, a : \vee A, x : U_e^o A; \Gamma$ ($[Tsh]$, (2) and (4))
(6) $\mathbf{take}\ x(a); \mathbf{share}\ x\{\mathbf{return}\ x(a); P||Q\} \vdash_\eta \Delta_1, \Delta_2, x : U_f^o A; \Gamma$ ($[Ttake]$, (5))
(7) $\mathbf{take}\ x(a); \mathbf{share}\ x\{\mathbf{return}\ x(a); P||Q\} \vdash_\eta \Delta; \Gamma$ ((1) and (6))

Case: $[RTsh]$ right-to-left

(1) $\Delta = \Delta_1, x = U_f^o A$
(2) $\mathbf{share}\ x\{\mathbf{return}\ x(a); P||Q) \vdash_\eta \Delta_1, a : \vee A, x : U_e^o A; \Gamma$, for some A, Δ_1
($[Ttake^{-1}$ and right side)
(3) $\Delta_1 = \Delta_{11}, \Delta_{12}$
(4) $\mathbf{return}\ x(a); P \vdash_\eta \Delta_{11}, x : U_f^o A; \Gamma$
(5) $Q \vdash_\eta \Delta_{12}, a : \vee A, x : U_e^c A; \Gamma$ ($[Tsh^{-1}]$ and (2))
(6) $\mathbf{take}\ x(a); Q \vdash_\eta \Delta_{12}, x : U_f^c A; \Gamma$ ($[Ttake]$, (5))
(7) $\mathbf{share}\ x\{\mathbf{return}\ x(a); P||\mathbf{take}\ x(a); Q\} \vdash_\eta \Delta_{11}, \Delta_{12}, x : U_f^o A; \Gamma$ ($[Tsh]$, (4) and (6))
(8) $\Delta_{11}, \Delta_{12}, x : U_f^o A = \Delta$ ((1) and (3))
(9) $\mathbf{share}\ x\{\mathbf{return}\ x(a); P||\mathbf{take}\ x(a); Q\} \vdash_\eta \Delta; \Gamma$ ((7) and (8))

Below is the proof extension for theorem 1(2). Similar to part 1, only new or affected rules need to be checked, while rule $[S_f U_f f]$ is removed entirely.

Case: $[S_f U_f t]$
The proof only needs to add the relation parameter. As the reduction is a cut directed at a cell, the state and usage types both are owners. Which concluded the proof.
Note: the same evaluation is applied to $[S_e U_e]$.

Case: $[S_f U_f f]$
$\mathbf{cut}\ \{\mathbf{cell}\ c(a.P)|c|\mathbf{return}\ c(a'); Q\} \rightarrow \mathbf{cut}\{P|a|\{a/a'\}Q\}$

(1) $\Delta = \Delta_1, \Delta_2$
(2) $\mathbf{cell}\ c(a.P) \vdash_\eta \Delta_1, c : S_f^o A; \Gamma$
(3) $\mathbf{return}\ c(a'); Q \vdash_\eta \Delta_2, c : U_f^o \overline{A}; \Gamma$
($[Tcut^{-1}]$ and left side)
(4) $P \vdash_\eta \Delta_1, a : \wedge A; \Gamma$
(5) $Q \vdash_\eta \Delta_2, a' : \vee \overline{A}; \Gamma$
(6) $\{a/a'\}Q \vdash_\eta \Delta_2, a : \vee \overline{A}; \Gamma$
(7) $\mathbf{cut}\ \{P|a : \wedge A|\{a/a'\}Q\} \vdash_\eta \Delta_1, \Delta_2; \Gamma$
(8) $\mathbf{cut}\ \{P|a : \wedge A|\{a/a'\}Q\} \vdash_\eta \Delta; \Gamma$

2.1.4 Progress

To ensure that the property of progress is preserved theorem 2 needs to be proven, which states that a process reduces if it is live. The proof comes from Lemma 4, which proves the property of liveness. The proof for this Lemma remains largely the same except for the newly added type rule $[Treturn]$ which needs to be accounted for. As this rule accounts for an action, a similar analysis is applied as for rule $[T1]$ as noted in the proof itself.

Lemma 4 depends on Lemma 3, which consists of 7 parts where the proof only needs to be extended for the first 3 parts which are about the combinations possible with share. These 3 cases need to be expanded to account for the added relation parameter, which means that each of these cases is split into 3 new sub-cases. Mainly:

1. With both usages being children;
2. With the left usage being the owner;
3. With the right usage being the owner.

Extension lemma 3(1)(1) Let $P \vdash \Delta, x : U_f^c A; \Gamma$ and $Q \vdash \Delta, x : U_f^c A; \Gamma$ be processes for which $P \downarrow_{x:act}$ and $Q \downarrow_{x:act}$. Then, **share** $x\{P||Q\}$.

This proof is the same as that of the original lemma 3(1) as no new cases are introduced (i.e. both are children so the return action is not available and no further restrictions apply).

Extension lemma 3(1)(2) Let $P \vdash \Delta, x : U_f^o A; \Gamma$ and $Q \vdash \Delta, x : U_f^c A; \Gamma$ be processes for which $P \downarrow_{x:act}$ and $Q \downarrow_{x:act}$. Then, **share** $x\{P||Q\}$.

The proof remains largely the same as that of the original lemma 3(1), but accounts for the additional reduction where return and take are competing.
Case: The root rule of both $P \downarrow_{x:act}$ and $Q \downarrow_{x:act}$ is $[act]$.
A new case is added where $\mathcal{A} =$ **return** $x(a); P'$ and $\mathcal{B} =$ **take** $x(a); Q'$.
By applying $[RTSh]$ we obtain
share $x\{$**return** $x(a); P'||$**take** $x(a); Q'\} \equiv$ **take** $x(a);$ **share** $x\{$**return** $x(a); P'||Q'\}$
Hence

$$\frac{\textbf{share } x\{\textbf{return } x(a); P'||\textbf{take } x(a); Q'\} \equiv \textbf{take } x(a); \textbf{share } x\{\textbf{return } x(a); P'||Q'\} \quad \dfrac{\textbf{take } x(y); R \downarrow_{x:act}}{s(\textbf{take } x(y); R) = x}}{\textbf{share } x\{P||Q\} \downarrow_{x:act}}$$

Extension lemma 3(1)(3) Let $P \vdash \Delta, x : U_f^c A; \Gamma$ and $Q \vdash \Delta, x : U_f^o A; \Gamma$ be processes for which $P \downarrow_{x:act}$ and $Q \downarrow_{x:act}$. Then, **share** $x\{P||Q\}$.
The proof is the same as that from Lemma 3(1)(2), following the same step as the original proof of Lemma 3(3) (i.e. exploiting the commutative property).

The proofs for the remaining lemmas 3(2)(1-3) remain the same as that of Lemma 3(2), mainly because the only path for progress is if one process fills the cell using the put action. As a consequence, the same holds for lemmas 3(3)(1-3) reusing the proof of Lemma 3(3).

2.1.5 Confluence

Confluence is proven for CLASS in theorem 3 also called the diamond property. The proof follows from Lemmas 11 and 5(3), which rely on Lemmas 6, 7, 8 and 9. Only two notes have to be made to ensure that the property of confluence is also proven for the modification of class. In lemma 6 the same analysis applies for [Treturn] as for [T0] as noted in the proof itself. For shares in lemma 9(1), the proof remains the same as the modification of share falls automatically in the case where no two takes are offered. For lemma 9(2) a new case is introduced for the rule [RTSh], however, the proof is the same as for [RSh] in the same way as noted in the case of [PSh]. Finally, lemma 9(3) adds the case for $[S_f U_f f]$ which is handled in the same way as the proof for $[1\bot]$, which proves the property of confluence for CLASS+.

2.2 Backwards-compatibility

The modification to CLASS is meant to expand the abilities of the language. Only one action is added to CLASS mainly return. Thus if return is ignored, will the modified CLASS essentially collapse to CLASS? To prove backwards compatibility it only needs to be shown that any process in CLASS can be translated into a process of CLASS+.

Theorem 1 There exists an encoding $[\![\cdot]\!] : \text{CLASS} \to \text{CLASS+}$.

Proof: The encoding can first be defined by the translation of the type and type rules, which can be translated one-to-one with the exception of the rules related to cells as these rules are the only ones modified. For the usage and state types, they can simply be encoded with the child relation (e.g. $[\![U_e]\!] = U_e^c$). As a consequence,

every rule that applies to cells except for [Tcell] and [Tempty] can be directly translated with the relation set to child. The encoding of [Tcell] and [Tempty] is a different case as it is not possible to set the relation parameter to child as it always expects an owner as its dual. So instead the following encoding is proposed:

$$\left[\!\!\left[\frac{P \vdash \Delta, a : \wedge A; \Gamma}{\textbf{cell } z(a.P) \vdash \Delta, z : S_f A; \Gamma}\right]\!\!\right] =$$

$$\dfrac{\dfrac{[\![P]\!] \vdash [\![\Delta]\!], a : \wedge[\![A]\!]; [\![\Gamma]\!]}{\textbf{cell } c(a.[\![P]\!]) \vdash [\![\Delta]\!], c : S_f^o[\![A]\!]; [\![\Gamma]\!]}\,Tcell \quad \dfrac{\dfrac{}{\textbf{discard } a \vdash a : \wedge\overline{[\![A]\!]}; [\![\Gamma]\!]}\,Tdiscard}{\textbf{return } c(a); \textbf{discard } ac \vdash c : U_f^o\overline{[\![A]\!]}; [\![\Gamma]\!]}\,Treturn \quad \dfrac{}{\textbf{fwd } c\ z \vdash c : U_f^c\overline{[\![A]\!]}, z : S_f^c[\![A]\!]; [\![\Gamma]\!]}\,Tfwd}{\textbf{share } c\{\textbf{return } c(a); \textbf{discard } a || \textbf{fwd } c\ z\} \vdash c : U_f^o\overline{[\![A]\!]}, z : S_f^c[\![A]\!]; [\![\Gamma]\!]}\,Tshare}{\textbf{cut } \{\textbf{cell } c(a.[\![P]\!]) | c | \textbf{share } c\{\textbf{return } c(a); \textbf{discard } a || \textbf{fwd } c\ z\}\} \vdash [\![\Delta]\!], z : S_f^c[\![A]\!]; [\![\Gamma]\!]}\,Tcut$$

$$\left[\!\!\left[\frac{}{\textbf{empty } z \vdash z : S_e A; \Gamma}\right]\!\!\right] =$$

$$\dfrac{\dfrac{}{\textbf{empty } c \vdash, c : S_e^o[\![A]\!]; [\![\Gamma]\!]}\,Tempty \quad \dfrac{\dfrac{\dfrac{}{\textbf{discard } a \vdash a : \wedge\overline{[\![A]\!]}; [\![\Gamma]\!]}\,Tdiscard}{\textbf{return } c(a); \textbf{discard } ac \vdash c : U_f^o\overline{[\![A]\!]}; [\![\Gamma]\!]}\,Treturn \quad \dfrac{}{\textbf{fwd } c\ z \vdash c : U_f^c\overline{[\![A]\!]}, z : S_f^c[\![A]\!]; [\![\Gamma]\!]}\,Tfwd}{\textbf{share } c\{\textbf{return } c(a); \textbf{discard } a || \textbf{fwd } c\ z\} \vdash c : U_f^o\overline{[\![A]\!]}, z : S_f^c[\![A]\!]; [\![\Gamma]\!]}\,Tshare}{\textbf{cut } \{\textbf{empty } c | c | \textbf{share } c\{\textbf{release } c || \textbf{fwd } c\ z\}\} \vdash [\![\Delta]\!], z : S_e^c[\![A]\!]; [\![\Gamma]\!]}\,Tcut$$

Essentially the encoding for these type rules adds a default owner process for the cell and forwards the state that corresponds with usage by children. It only remains to be verified that the operational semantics remain the same by checking soundness and completeness.

For the reduction rules that do not involve a cell the semantics remain the same, which means that it satisfies both the properties of completeness and soundness. For the remain rules $[S_f U_f t]$, $[S_e U_e]$ and $[S_f U_f f]$ the properties need to be proven.

Starting with $[S_f U_f t]$ completeness can be shown as follows, with the support of the congruence rules $[CC]$ and $[CSh]$ and reduction rule $[fwd]$:

$$\begin{aligned}
&\textbf{cut } \{\textbf{cut } \{\textbf{cell } c(a.P) | c | \textbf{share } c\{\textbf{return } c(a); \textbf{discard } a || \textbf{fwd } c\ z\}\} | z | \textbf{take } z(a'); Q\} \equiv \\
&\textbf{cut } \{\textbf{cell } c(a.P) | c | \textbf{share } c\{\textbf{return } c(a); \textbf{discard } a || \textbf{cut } \{\textbf{fwd } c\ z | z | \textbf{take } z(a'); Q\}\}\} \rightarrow \\
&\textbf{cut } \{\textbf{cell } c(a.P) | c | \textbf{share } c\{\textbf{return } c(a); \textbf{discard } a || \textbf{take } z(a'); Q\} \equiv \\
&\textbf{cut } \{\textbf{cell } c(a.P) | c | \textbf{take } z(a'); \textbf{share } c\{\textbf{return } c(a); \textbf{discard } a || Q\}
\end{aligned}$$

This then allows us to apply the rule $[S_f U_f t]$in CLASS+ to be applied. As there are not other reductions possible it can only be concluded that the translation for this rule is also complete. For $[S_e U_e]$ a similar analysis holds. For rule $[S_f U_f f]$ the initial steps from the proof can be followed to reduce cut on the fwd. This is then followed by the congruence rule $[RSh]$ and the reduction $[S_f U_f f]$

$$\begin{aligned}
&\textbf{cut } \{\textbf{cut } \{\textbf{cell } c(a.P) | c | \textbf{share } c\{\textbf{return } c(a); \textbf{discard } a || \textbf{fwd } c\ z\}\} | z | \textbf{take } z(a'); Q\} \equiv \\
&\textbf{cut } \{\textbf{cell } c(a.P) | c | \textbf{share } c\{\textbf{return } c(a); \textbf{discard } a || \textbf{cut } \{\textbf{fwd } c\ z | z | \textbf{release } z\}\}\} \rightarrow \\
&\textbf{cut } \{\textbf{cell } c(a.P) | c | \textbf{share } c\{\textbf{return } c(a); \textbf{discard } a || \textbf{release } z\} \equiv \\
&\textbf{cut } \{\textbf{cell } c(a.P) | c | \textbf{return } c(a); \textbf{discard } a\} \rightarrow \\
&\textbf{cut } \{P | a | \textbf{discard } a\}
\end{aligned}$$

This shows that the property holds for completeness, and as there are no other reductions for the encoded process the property of soundness holds as well. This also means that for the encoding the properties hold and this concludes that there is an encoding from CLASS to CLASS+.

3 Encoding

The encoding from λ_{lock} to CLASS+ can now be constructed. First, the encoding is defined for the types as it is a prerequisite for the encoding of type rules which follow. Finally, as the semantics of a λ_{lock} process is given as a set of threads and locks the encoding is extended such that the preservation of reduction in the translation is verifiable.

3.1 Encoding of Types

The encoding of types is given in fig. 1. Each type has a direct equivalent in CLASS+ however there are some considerations. The first consideration is the use of channels and duals. A similar approach is followed as done in the translation of GV into CP by Walder [5], where inputs are considered dual of the type and outputs are considered simply the encoding. Furthermore, as CLASS+ uses affine channels, in particular for stored processes in cells, this needs to be properly captured in order to ensure that it is always possible to make any encoded process affine. In particular, a process with some channel can only be made affine if all other channels are either using an affine channel (coaffine) or a cell (usage). Hence, for inputs the type needs to be not only dual but also coaffine. It is not necessary to make a distinction between affine channels or usages of cells as the latter can simply be transformed into an affine channel without any repercussions.

For the encoding of lock types to cell types it is chosen to use the state type as opposed to the usage type. The state type allows an encoded process to be used with a cut, and forward it as an output as necessary. For the remaining part, which are the type of content, the ownership and the state of the cell, there is a one-to-one correspondence between sub-types of locks and cells. Finally, it also has to be noted that the sub-types of sum and product types always need to be affine, as it is not possible to make them affine later as their application requires them to be cut directly against their counterparts which uses inputs which were made coaffine and dual.

The recursive type was dropped as it falls outside the scope of this project considering that there is no associated type rule. Similarly, the type **0** was dropped with its application in match, as it was unclear what the author intended and might only apply in order to deal with correctness of types or type preservation.

$$
\begin{array}{rcl}
[\![\mathbf{1}]\!] & = & \mathbf{1} \\
[\![\tau_1 + \tau_2]\!] & = & (\wedge[\![\tau_1]\!]) \oplus (\wedge[\![\tau_2]\!]) \\
[\![\tau_1 \times \tau_2]\!] & = & (\wedge[\![\tau_1]\!]) \otimes (\wedge[\![\tau_2]\!]) \\
[\![\tau_1 \multimap \tau_2]\!] & = & (\vee\overline{[\![\tau_1]\!]}) \parr [\![\tau_2]\!] \\
[\![\tau_1 \to \tau_2]\!] & = & !(\vee\overline{[\![\tau_1]\!]} \parr [\![\tau_2]\!]) \\
[\![\mathbf{Lock}\langle \tau^{a=a_1+a_2}_{b=b_1+b_2} \rangle]\!] & = & \mathbf{S}^{[\![a]\!]=[\![a_1]\!]+[\![a_2]\!]}_{[\![b]\!]=[\![b_1]\!]+[\![b_2]\!]}\ [\![\tau]\!] \\
 & & \text{where } a, a_1, a_2 \in \mathcal{R} \text{ and } b, b_1, b_2 \in \mathcal{S} \\
[\![a]\!] & = & \begin{cases} o & \textit{if } a = 1 \\ c & \textit{otherwise} \end{cases} \\
[\![b]\!] & = & \begin{cases} e & \textit{if } b = 1 \\ f & \textit{otherwise} \end{cases}
\end{array}
$$

Figure 1: Encoding of types

3.2 Encoding of Type Rules

The encoding of the type rules is given in figs. 2 to 4. In the encoding of the types, considerations were made with respect to the construction of the encoding of the type rules. Encoding of the expressions all follows a passing style, where one channel is reserved as output. Each of the encodings can use multiple type rules in CLASS+ to achieve the same result, but they end in the same preconditions as the original type rule. The encoded type rules also translate the set of arguments to a set of channels, each of these is always dual and coaffine as mentioned before to ensure that the output channel can always be made affine. It is not always necessary to make an output affine hence, outputs are not affine by default which reduces the work of the encoding and the proofs that will follow. In the encoded processes, the channel sets are translated as a whole, where the context differentiates whether the original argument was linear (and on the left side of the semi-colon) or exponential (and on the right side). The result of these encodings is both to translate expressions as well as to ensure type preservation and correctness of the resulting encoded process in CLASS+.

Theorem 2: If $\Gamma \vdash e : \tau$ then $[\![e]\!]_z \vdash_\eta \vee\overline{[\![\Gamma]\!]}, z : [\![\tau]\!]; \overline{[\![\Gamma]\!]}$.

Proof: The encoded type rules are given in figs. 2 to 4.

The encoded type rules for new simply create a new lock of the encoded type, the choice for translation of lock type to that of state also becomes apparent as it can be used directly with a cut to an encoded thread. The

drop action would semantically be the same as release, though release does not produce an output type and hence a parallel composition is used to make it type-compatible. The wait action is directly compatible with return, but needs to account for the affine resource and removes this property before it forwards it to the ouput channel. Fork is similar to share, however, it constricts in how a new thread is created by using a function that the lock can be passed onto while also making terminate. Hence, in this translation, the new thread process is encoded as Q as a function call which also handles the termination. Acquire and release are translated quite directly with their counterparts take and put respectively, but with the additional consideration of removing the affine property through use and making the stored resource affine first.

Nothing needs to be said about the encoding of termination as it has its direct counterpart close acting on the output channel. For encoding of an argument, the requirement to keep inputs coaffine such that any process can be stored inside a cell while the output must not be. To achieve this using the affine input gives us a forward that does not need to be affine, while retaining the possibility to make it so. Function and a function call is encoded through recv and send, where the call performs the cut between the two encoded processes. As the input of the function is coaffine, it is necessary to make the output channel of the input affine. The exponential function and call are translated in the same way, but are wrapped using ! and ? + call as to make the channel exponential and make the necessary reductions possible. The encodings of a tuple is simply done by combining the outputs through a send action, but not before making the channels affine this as in the encoding for let the variables are encoded as coaffine. The let encoding performs the cut and does the opposite of a tuple by using recv. Encodings for in L and R are very similar to that of a tuple, but with one process. Their counterpart match is encoded as a consequence the same as let, but without the recv as there is only one channel as input.

3.3 Encoding of the Process

As the operational semantics are given as a set of threads and locks that reduce an additional layer of translation is necessary. Essentially, it will allow a program and its translation to be verified from any state, not only from the start and the end. This part of the encoding works off of the fundamental definition of references to locks and shares of cells. The definition of locks tells us that between threads at most they can have one lock in common. Similarly, this definition holds also true for shares of usages of cells in CLASS+ per the definition of the type rules and preservation of types. Furthermore, as locks have an owner, there must always be at least one thread for each lock. For threads, it does not hold true that they need a reference to a lock, as a thread is always allowed to drop any reference leaving an independent thread behind. These assertions form the basis of the encoding of the process.

The encoding from the previous sections can be used with the extension of encoding of threads, locks and references. In this encoding a special case is considered when a process terminates and is of $\mathbf{1}$, as to account for the semantic rule exit in λ_{lock} where the thread terminates.

$$
\begin{aligned}
[\![k \mapsto \text{Lock}(c, \text{None}))]\!]_k &= \mathbf{empty}\ k \\
[\![k \mapsto \text{Lock}(c, \text{Some}(v)))]\!]_k &= \mathbf{cell}\ k(a.\mathbf{affine}_{c,a}\ a; [\![v]\!]_a) \\
[\![n \mapsto \text{Thread}(e)]\!]_n &= \begin{cases} [\![e]\!]_n & \text{if } \tau_n \neq \mathbf{1} \\ \mathbf{cut}\ \{[\![e]\!]_x\ |x : \mathbf{1}|\ \mathbf{wait}\ x;\ \mathbf{0}\} & \text{otherwise} \end{cases} \\
[\![\langle k \rangle]\!]_z &= \mathbf{fwd}\ k\ z
\end{aligned}
$$

Definition 1: If $\rho \in \text{Cfg}$ then $(\!|\rho|\!)$ which is the encoding for the process.

ρ adheres to the defined principle of λ_{lock}, meaning it is a directed acyclic graph of threads linked to locks. This means that it can be written as a disjoint union, with any thread not referencing any lock falling into the set ρ'.

$$\rho = \rho_1 \sqcup ... \sqcup \rho_n \sqcup \rho'$$

This can be encoded as a base case where ρ consists of multiple disjoint DAGs. This is possible as there are no common channels in the encoded threads.

$$(\!|\rho|\!) = \mathbf{par}\ \{(\!|\rho_1|\!) \parallel ... \parallel (\!|\rho_n|\!) \parallel Q'\} \text{ where } Q' = \mathbf{par}\ \{\|_{t_i \in \rho'}\ [\![t]\!]_i\}$$

This leaves the case where ρ is not disjoint. The process is built inductively by selecting a lock and threads that reference it. For the choice, locks that hold another lock take priority, as they must be inside the cut of the lock they are holding. This is also done for the inductive step, counting on the property that there will always be one lock without holding a reference (empty locks never hold a reference).

$$\begin{aligned} &l \in \rho_{\text{locks}} \\ &T = \{t \in \rho_{\text{threads}} | t \text{ references } l\} \\ &\rho' = \rho \setminus T \cup \{l\} \end{aligned}$$

By definition of λ_{lock} each thread has at most one lock in common, hence each encoded thread has only reference to l. This can be used to create a share that cuts on the encoded lock.

$$Q = \textbf{cut } \{[\![l]\!]_x \ | x : \mathbf{S}^o_{[\![b]\!]} \ [\![\tau_l]\!] | \textbf{ share } x \ \{\|_{t_i \in T} \ [\![t_i]\!]_{t_i}\}\}$$

The inductive step is where the process Q is extended by selecting another lock one that is also used by Q. As our criteria for ρ was that it is a DAG a natural consequence is that some thread used for the construction of Q also references lock other than l. Hence a new lock is selected and a set of threads is constructed. Here ρ' becomes ρ.

$$\begin{aligned} &l' \in \rho \text{ where } \exists t \in T : t \text{ references } l' \\ &T' = \{t \in \rho_{\text{threads}} | t \text{ references } l\} \\ &\rho' = \rho \setminus T' \cup \{l'\} \end{aligned}$$

Then the process Q can be extended.

$$Q' = \textbf{cut } \{[\![l']\!]_x \ | x : \mathbf{S}^o_{[\![b]\!]} \ [\![\tau_{l'}]\!] | \textbf{ share } x \ \{\|_{t_i \in T'} \ [\![t_i]\!]_{t_i} \parallel Q\}\}$$

The inductive step is repeated with Q' becoming Q and ρ' becoming ρ, until there are no more locks left. By definition of λ_{lock} any lock always has at least one thread referencing it, so T and T' is never empty, while because of the choice for ρ being a DAG there are no threads left once there are no more locks. Hence, the set will be empty.

As the process ρ is an unordered set and no assumptions are made on the order of the references it needs to be verified that the order in which the shares are constructed does not matter. The proof follows from the distributive conversions as defined in CLASS+. By congruence rule $[ShSh]$ it is possible to swap shares, hence choosing to use one lock over the other first for construction does not matter.

This section directly relates to the operational semantics of λ_{lock} and it remains to verify the preservation of the semantics of the encoding.

3.3.1 Encoding of the examples

In the introduction some examples of λ_{lock} were given in this part these will be translated as a verification of soundness and completeness of the encoding in these examples.

$$\rho = \left\{ \begin{array}{c} t_1 \mapsto \text{Thread}(K_1[\textbf{let } x, y = \textbf{acquire}(\langle k \rangle) \textbf{ in release}(x, 2 * y)]) \\ t_2 \mapsto \text{Thread}(K_2[\textbf{let } x, y = \textbf{acquire}(\langle k \rangle) \textbf{ in release}(x, 2 + y)]) \\ k \mapsto \text{Lock}(1, \text{Some}(1))) \end{array} \right\} \rightsquigarrow^*$$

$$\rho'_1 = \left\{ \begin{array}{c} \dots \\ k \mapsto \text{Lock}(1, \text{Some}(4))) \end{array} \right\} \textit{ or}$$

$$\rho'_2 = \left\{ \begin{array}{c} \dots \\ k \mapsto \text{Lock}(1, \text{Some}(6))) \end{array} \right\}$$

This process can be encoded as (the encoding of pairs are omitted for clarity):

$$
\begin{aligned}
(\!|\rho|\!) &= \mathbf{cut}\ \{\mathbf{cell}\ x(y.1)\ |x : \mathbf{S}^o_f\ int|\ \mathbf{share}\ x\ \{\mathbf{take}\ x(y); \mathbf{put}\ x(y.2 * y); [\![K_1]\!]_{t_1}\ \|\ \mathbf{take}\ x(y); \mathbf{put}\ x(y.2 + y); [\![K_2]\!]_{t_2}\}\} \\
&\equiv \mathbf{cut}\ \{\mathbf{cell}\ x(y.1)\ |x : \mathbf{S}^o_f\ int|\ \mathbf{take}\ x(y); \mathbf{share}\ x\ \{\mathbf{put}\ x(y.2 * y); [\![K_1]\!]_{t_1}\ \|\ \mathbf{take}\ x(y); \mathbf{put}\ x(y.2 + y); [\![K_2]\!]_{t_2}\}\} \\
&\quad + \mathbf{cut}\ \{\mathbf{cell}\ x(y.4)\ |x : \mathbf{S}^o_f\ int|\ \mathbf{take}\ x(y); \mathbf{share}\ x\ \{\mathbf{take}\ x(y); \mathbf{put}\ x(y.2 * y); [\![K_1]\!]_{t_1}\ \|\ \mathbf{put}\ x(y.2 + y); [\![K_2]\!]_{t_2}\}\} \\
&\rightsquigarrow \mathbf{cut}\ \{\mathbf{cell}\ x(y.4)\ |x : \mathbf{S}^o_f\ int|\ \mathbf{share}\ x\ \{[\![K_1]\!]_{t_1}\ \|\ [\![K_2]\!]_{t_2}\}\} \\
&\quad + \mathbf{cut}\ \{\mathbf{cell}\ x(y.6)\ |x : \mathbf{S}^o_f\ int|\ \mathbf{share}\ x\ \{[\![K_1]\!]_{t_1}\ \|\ [\![K_2]\!]_{t_2}\}\} + ... \\
&\equiv (\!|\rho'_1|\!) + (\!|\rho'_2|\!) + ...
\end{aligned}
$$

This example demonstrates the confluent property as the process reduces to both possible outcomes that can come from the non-deterministic competition between threads in acquiring/taking values from locks/cells. In the verification of completeness, a possible reduction in the original process encoded is only a single item in the sum of outcomes, without knowing the other outcomes. Soundness is verified by checking that all outcomes have some reduction for the original process. In the encoded example, there are the triple dots, this was included as in the encoding it is unsure what K_1 and K_2 might do. The options remain open to release, return, or even to take. In the latter case, this would result in additional competition that leads to different outcomes. As it is unknown, for the purposes of this example no assumptions are made.

The other example of lock passing can be encoded in two ways which are both equivalent as shown below.

$$
\begin{aligned}
&\mathbf{cut}\ \{\mathbf{empty}\ k_1\ |k_1 : \mathbf{S}^o_e\ [\![\tau_{k_1}]\!]|\ \mathbf{share}\ k_1\ \{\mathbf{take}\ k_1(x); \mathbf{use}\ x; k_2[\![K_1]\!]_{t_2} \\
&\ \|\ \mathbf{cut}\ \{\mathbf{share}\ k_2\ \{\mathbf{put}\ k_1(x.\mathbf{affine}_{c,a}\ x; \mathbf{fwd}\ k_2\ x); [\![K_1]\!]_{t_1}\ \|\ [\![t_3]\!]_{t_3}\}\ |k_2 : \mathbf{S}^o_{[\![b]\!]}\ [\![\tau_{k_2}]\!]|\ \}\}\} \\
\equiv\ &\mathbf{cut}\ \{[\![k_2]\!]_{k_2}\ |k_2 : \mathbf{S}^o_{[\![b]\!]}\ [\![\tau_{k_2}]\!]|\ \mathbf{share}\ k_2\ \{[\![t_3]\!]_{t_3} \\
&\ \|\ \mathbf{cut}\ \{\mathbf{empty}\ k_1\ |k_1 : \mathbf{S}^o_e\ [\![\tau_{k_1}]\!]|\ \mathbf{share}\ k_1\ \{\mathbf{put}\ k_1(x.\mathbf{affine}_{c,a}\ x; \mathbf{fwd}\ k_2\ x); [\![K_1]\!]_{t_1}\ \|\ \mathbf{take}\ k_1(x); [\![K_1]\!]_{t_2}\}\}\} \\
\rightsquigarrow\ &\mathbf{cut}\ \{[\![k_2]\!]_{k_2}\ |k_2 : \mathbf{S}^o_{[\![b]\!]}\ [\![\tau_{k_2}]\!]|\ \mathbf{share}\ k_2\ \{[\![t_3]\!]_{t_3} \\
&\ \|\ \mathbf{cut}\ \{\mathbf{cell}\ k_1(k_2.[\![k_2]\!]_{k_2})\ |k_1 : \mathbf{S}^o_f\ [\![\tau_{k_1}]\!]|\ \mathbf{share}\ k_1\ \{[\![K_1]\!]_{t_1}\ \|\ \mathbf{take}\ k_1(x); [\![K_1]\!]_{t_2}\}\}\}
\end{aligned}
$$

In the encodings, the choice for which lock to encode first does not matter as both are equivalent, both before and after the reduction. After the first reduction has taken place, it will be possible for thread t_2 to receive the reference to lock. Here it becomes obvious why prioritize locks that hold reference to some other lock so they fall inside the appropriate cut. Depending on the context K_1, the next reduction would be where thread t_2 receives the reference stored in the lock.

4 Future work

While the encoding is given and the operation semantics are considered to be preserved, it still needs to be formally proven that the encoding is both sound and complete. Without these properties, it is possible that the encoded process does not yield the same results as the original process, while it is also possible that it does more than it was supposed to.

The extension of λ_{lock} that uses lock groups and partial orders to share multiple locks and ensure deadlock freedom is not incorporated as it requires a further extension of CLASS. Adding partial orders would allow for the extension of the encoding to translate expressions and produce a complete set of outcomes as a consequence of confluence.

5 Conclusion

In this project an encoding from λ_{lock} to CLASS+ has been developed. The encoding introduces confluence to an otherwise non-confluent calculus, creating transparency into the possible outcomes of said program. The encoding follows multiple steps, starting from the encoding of types which need to account for not only the duality of channels but also to ensure it is always possible to make a channel affine. The type rules are encoded to encoded expressions that are ensured to preserve types. Finally, as the semantics are given as unordered sets of locks and threads an encoding was developed for these. It follows the properties as defined by λ_{lock} as a base assumption on the shape of the sharing topology, which is also necessary as CLASS+ uses these same definitions.

In order to facilitate the encoding, it was deemed necessary to expand CLASS into CLASS+ due to differences in how the end of lifecycles of locks and cells worked. Where locks would preserve the values, cells would

simply discard them without a way to deviate from this position. In order to overcome this the same notion of ownership present in λ_{lock} was added to CLASS. The addition of a return operator allows for the desired behaviour. Important properties were then verified like, type preservation, progress and confluence.

The encoding shows that λ_{lock} and CLASS+ fulfil the same purpose. However, where λ_{lock} discards confluence and CLASS+ preserves it. Thus the encoding generates a degree of transparency into the outcomes of any program.

6 Acknowledgements

I would like to thank my supervisor Jorge Pérez for guiding me through this internship project. I found it an interesting project and with his guidance, I learned new aspects of λ-calculus and π-calculus and their similarities.

References

[1] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *International Conference on Concurrency Theory*, pages 222–236. Springer, 2010.

[2] Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 26(3):367–423, 2016.

[3] Jules Jacobs and Stephanie Balzer. Higher-order leak and deadlock free locks. *Proc. ACM Program. Lang.*, 7(POPL), jan 2023.

[4] PEDRO MANUEL SABINO ROCHA. Class: A logical foundation for typeful programming with shared state. 2022.

[5] Philip Wadler. Propositions as sessions. *SIGPLAN Not.*, 47(9):273–286, sep 2012.

$$\left[\!\left[\frac{\Gamma \text{ unr}}{\Gamma \vdash \mathbf{new}() : \mathbf{Lock}\langle \tau_1^1 \rangle} \right]\!\right]_z = \frac{}{\mathbf{empty}\ z \vdash_\eta z : \mathbf{S}_e^o\ [\![\tau]\!]; \overline{[\![\Gamma]\!]}}\ Tempty$$

$$\left[\!\left[\frac{\Gamma \vdash e : \mathbf{Lock}\langle \tau_0^0 \rangle}{\Gamma \vdash \mathbf{drop}(e) : \mathbf{1}} \right]\!\right]_z =$$

$$\frac{[\![e]\!]_x \vdash_\eta \vee[\![\Gamma]\!], x : \mathbf{S}_f^c\ [\![\tau]\!]; \overline{[\![\Gamma]\!]} \quad \dfrac{\dfrac{}{\mathbf{release}\ x \vdash_\eta x : \mathbf{U}_f^c\ \overline{[\![\tau]\!]}; \overline{[\![\Gamma]\!]}}\ Trelease \quad \dfrac{}{\mathbf{close}\ z \vdash_\eta z : \mathbf{1}; \overline{[\![\Gamma]\!]}}\ T1}{\mathbf{par}\ \{\mathbf{release}\ x \parallel \mathbf{close}\ z\} \vdash_\eta x : \mathbf{U}_f^c\ \overline{[\![\tau]\!]}, z : \mathbf{1}; \overline{[\![\Gamma]\!]}}\ Tmix}{\mathbf{cut}\ \{[\![e]\!]_x\ |x : \mathbf{S}_f^c\ [\![\tau]\!]|\ \mathbf{par}\ \{\mathbf{release}\ x \parallel \mathbf{close}\ z\}\} \vdash_\eta \vee[\![\Gamma]\!], z : \mathbf{S}_e^o\ [\![\tau]\!]; \overline{[\![\Gamma]\!]}}\ Tcut$$

$$\left[\!\left[\frac{\Gamma \vdash e : \mathbf{Lock}\langle \tau_0^1 \rangle}{\Gamma \vdash \mathbf{wait}(e) : \tau} \right]\!\right]_z =$$

$$\frac{[\![e]\!]_x \vdash_\eta \vee[\![\Gamma]\!], x : \mathbf{S}_f^o\ [\![\tau]\!]; \overline{[\![\Gamma]\!]} \quad \dfrac{\dfrac{\dfrac{}{\mathbf{fwd}\ a\ z \vdash_\eta a : \overline{[\![\tau]\!]}, z : [\![\tau]\!]; \overline{[\![\Gamma]\!]}}\ Tfwd}{\mathbf{use}\ a; \mathbf{fwd}\ a\ z \vdash_\eta a : \vee\overline{[\![\tau]\!]}, z : [\![\tau]\!]; \overline{[\![\Gamma]\!]}}\ Tuse}{\mathbf{return}\ x(a); \mathbf{use}\ a; \mathbf{fwd}\ a\ z \vdash_\eta x : \mathbf{U}_f^o\ \overline{[\![\tau]\!]}, z : [\![\tau]\!]; \overline{[\![\Gamma]\!]}}\ Treturn}{\mathbf{cut}\ \{[\![e]\!]_x\ |x : \mathbf{S}_f^o\ [\![\tau]\!]|\ \mathbf{return}\ x(a); \mathbf{use}\ a; \mathbf{fwd}\ a\ z\} \vdash_\eta \vee[\![\Gamma]\!], z : [\![\tau]\!]; \overline{[\![\Gamma]\!]}}\ Tcut$$

$$\left[\!\left[\frac{\Gamma \equiv \Gamma_1 \cdot \Gamma_2 \quad \Gamma_1 \vdash e_1 : \mathbf{Lock}\langle \tau_{b_1+b_2}^{a_1+a_2} \rangle \quad \Gamma_2 \vdash e_2 : \mathbf{Lock}\langle \tau_{b_2}^{a_2} \rangle \multimap 1}{\Gamma \vdash \mathbf{fork}(e_1, e_2) : \mathbf{Lock}\langle \tau_{b_1}^{a_1} \rangle} \right]\!\right]_z =$$

$$\frac{[\![e_1]\!]_x \vdash_\eta \vee[\![\Gamma_1]\!], x : \mathbf{S}_{[\![b_1]\!]+[\![b_2]\!]}^{[\![a_1]\!]+[\![a_2]\!]}\ [\![\tau]\!]; \overline{[\![\Gamma]\!]} \quad \dfrac{\dfrac{}{\mathbf{fwd}\ x\ z \vdash_\eta x : \mathbf{U}_{[\![b_1]\!]}^{[\![a_1]\!]}\ \overline{[\![\tau]\!]}, z : \mathbf{S}_{[\![b_1]\!]}^{[\![a_1]\!]}\ [\![\tau]\!]; \overline{[\![\Gamma]\!]}}\ Tfwd \quad Q}{\mathbf{share}\ x\ \{\mathbf{fwd}\ x\ z \parallel Q\} \vdash_\eta \vee[\![\Gamma_2]\!], x : \mathbf{U}_{[\![b_1]\!]+[\![b_2]\!]}^{[\![a_1]\!]+[\![a_2]\!]}\ \overline{[\![\tau]\!]}, z : \mathbf{S}_{[\![b_1]\!]}^{[\![a_1]\!]}\ [\![\tau]\!]; \overline{[\![\Gamma]\!]}}\ Tshare}{\mathbf{cut}\ \{[\![e_1]\!]_x\ |x : \mathbf{S}_{[\![b_1]\!]+[\![b_2]\!]}^{[\![a_1]\!]+[\![a_2]\!]}\ [\![\tau]\!]|\ \mathbf{share}\ x\ \{\mathbf{fwd}\ x\ z \parallel Q\}\} \vdash_\eta \vee[\![\Gamma]\!], z : \mathbf{S}_{[\![b_1]\!]}^{[\![a_1]\!]}\ [\![\tau]\!]; \overline{[\![\Gamma]\!]}}\ Tcut$$

with

$$\frac{[\![e_2]\!]_y \vdash_\eta [\![\Gamma]\!], y : \mathbf{U}_{[\![b_2]\!]}^{[\![a_2]\!]}\ \overline{[\![\tau]\!]}\ \parr\ \mathbf{1}; \overline{[\![\Gamma]\!]} \quad \dfrac{\dfrac{}{\mathbf{fwd}\ x'\ x \vdash_\eta x' : \mathbf{S}_{[\![b_2]\!]}^{[\![a_2]\!]}\ [\![\tau]\!], x : \mathbf{U}_{[\![b_2]\!]}^{[\![a_2]\!]}\ \overline{[\![\tau]\!]}; \overline{[\![\Gamma]\!]}}\ Tfwd \quad \dfrac{\dfrac{}{\mathbf{0} \vdash_\eta \emptyset; \overline{[\![\Gamma]\!]}}\ T0}{\mathbf{wait}\ y;\ \mathbf{0} \vdash_\eta y : \bot; \overline{[\![\Gamma]\!]}}\ T\bot}{\mathbf{send}\ y(x'.\mathbf{fwd}\ x'\ x); \mathbf{wait}\ y;\ \mathbf{0} \vdash_\eta x : \mathbf{U}_{[\![b_2]\!]}^{[\![a_2]\!]}\ \overline{[\![\tau]\!]}, y : \mathbf{S}_{[\![b_2]\!]}^{[\![a_2]\!]}\ [\![\tau]\!] \oplus \bot; \overline{[\![\Gamma]\!]}}\ Tsend}{Q = \mathbf{cut}\ \{[\![e_2]\!]_y\ |y : \mathbf{U}_{[\![b_2]\!]}^{[\![a_2]\!]}\ \overline{[\![\tau]\!]}\ \parr\ \mathbf{1}|\ \mathbf{send}\ y(x'.\mathbf{fwd}\ x'\ x); \mathbf{wait}\ y;\ \mathbf{0}\} \vdash_\eta x : \mathbf{U}_{[\![b_2]\!]}^{[\![a_2]\!]}\ \overline{[\![\tau]\!]}; \overline{[\![\Gamma]\!]}}\ Tcut$$

$$\left[\!\left[\frac{\Gamma \vdash e : \mathbf{Lock}\langle \tau_0^a \rangle}{\Gamma \vdash \mathbf{acquire}(e) : \mathbf{Lock}\langle \tau_1^a \rangle} \right]\!\right]_z =$$

$$\frac{[\![e]\!]_x \vdash_\eta \vee[\![\Gamma]\!], x : \mathbf{S}_f^{[\![a]\!]}\ [\![\tau]\!]; \overline{[\![\Gamma]\!]} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\mathbf{fwd}\ x'\ x \vdash_\eta x' : \mathbf{S}_e^{[\![a]\!]}\ [\![\tau]\!], x : \mathbf{U}_e^{[\![a]\!]}\ \overline{[\![\tau]\!]}; \overline{[\![\Gamma]\!]}}\ Tfwd \quad \dfrac{}{\mathbf{fwd}\ a\ z \vdash_\eta a : \overline{[\![\tau]\!]}, z : [\![\tau]\!]; \overline{[\![\Gamma]\!]}}\ Tfwd}{\mathbf{send}\ z(x'.\mathbf{fwd}\ x'\ x); \mathbf{fwd}\ a\ z \vdash_\eta x : \mathbf{U}_e^{[\![a]\!]}\ \overline{[\![\tau]\!]}, a : \overline{[\![\tau]\!]}, z : \mathbf{S}_e^{[\![a]\!]}\ [\![\tau]\!] \otimes [\![\tau]\!]; \overline{[\![\Gamma]\!]}}\ Tsend}{\mathbf{use}\ a; \mathbf{send}\ z(x'.\mathbf{fwd}\ x'\ x); \mathbf{fwd}\ a\ z \vdash_\eta x : \mathbf{U}_e^{[\![a]\!]}\ \overline{[\![\tau]\!]}, a : \vee\overline{[\![\tau]\!]}, z : \mathbf{S}_e^{[\![a]\!]}\ [\![\tau]\!] \otimes [\![\tau]\!]; \overline{[\![\Gamma]\!]}}\ Tuse}{\mathbf{take}\ x(a); \mathbf{use}\ a; \mathbf{send}\ z(x'.\mathbf{fwd}\ x'\ x); \mathbf{fwd}\ a\ z \vdash_\eta x : \mathbf{U}_f^{[\![a]\!]}\ \overline{[\![\tau]\!]}, z : \mathbf{S}_e^{[\![a]\!]}\ [\![\tau]\!] \otimes [\![\tau]\!]; \overline{[\![\Gamma]\!]}}\ Ttake}{\mathbf{cut}\ \{[\![e]\!]_x\ |x : \mathbf{S}_f^{[\![a]\!]}\ [\![\tau]\!]|\ \mathbf{take}\ x(a); \mathbf{use}\ a; \mathbf{send}\ z(x'.\mathbf{fwd}\ x'\ x); \mathbf{fwd}\ a\ z\} \vdash_\eta \vee[\![\Gamma]\!], z : \mathbf{S}_e^{[\![a]\!]}\ [\![\tau]\!] \otimes [\![\tau]\!]; \overline{[\![\Gamma]\!]}}\ Tcut$$

$$\left[\!\left[\frac{\Gamma \equiv \Gamma_1 \cdot \Gamma_2 \quad \Gamma_1 \vdash e_1 : \mathbf{Lock}\langle \tau_1^a \rangle \quad \Gamma_2 \vdash e_2 : \tau}{\Gamma \vdash \mathbf{release}(e_1, e_2) : \mathbf{Lock}\langle \tau_0^a \rangle} \right]\!\right]_z =$$

$$\frac{[\![e_1]\!]_x \vdash_\eta \vee[\![\Gamma_1]\!], x : \mathbf{S}_e^{[\![a]\!]}\ [\![\tau]\!]; \overline{[\![\Gamma]\!]} \quad \dfrac{\dfrac{[\![e_2]\!]_a \vdash_\eta \vee[\![\Gamma_2]\!], a : [\![\tau]\!]; \overline{[\![\Gamma]\!]}}{\mathbf{affine}_{c,a}\ a; [\![e_2]\!]_a \vdash_\eta \vee[\![\Gamma_2]\!], a : \wedge[\![\tau]\!]; \overline{[\![\Gamma]\!]}}\ Taffine \quad \dfrac{}{\mathbf{fwd}\ x\ z \vdash_\eta x : \mathbf{U}_f^{[\![a]\!]}\ \overline{[\![\tau]\!]}, z : \mathbf{S}_f^{[\![a]\!]}\ [\![\tau]\!]; \overline{[\![\Gamma]\!]}}\ Tfwd}{\mathbf{put}\ x(a.[\![e_2]\!]_a); \mathbf{fwd}\ x\ z \vdash_\eta \vee[\![\Gamma_2]\!], x : \mathbf{U}_e^{[\![a]\!]}\ \overline{[\![\tau]\!]}, z : \mathbf{S}_f^{[\![a]\!]}\ [\![\tau]\!]; \overline{[\![\Gamma]\!]}}\ Tput}{\mathbf{cut}\ \{[\![e_1]\!]_x\ |x : \mathbf{S}_e^{[\![a]\!]}\ [\![\tau]\!]|\ \mathbf{put}\ x(a.[\![e_2]\!]_a); \mathbf{fwd}\ x\ z\} \vdash_\eta \vee[\![\Gamma]\!], z : \mathbf{S}_f^{[\![a]\!]}\ [\![\tau]\!]; \overline{[\![\Gamma]\!]}}\ Tcut$$

Figure 2: Encoding of type rules of lock interactions

$$\left[\!\left[\frac{\Gamma \text{ unr}}{\Gamma \ \vdash \ () \ : \ \mathbf{1}}\right]\!\right]_z = \frac{}{\mathbf{close}\ z \vdash_\eta z : \mathbf{1}; \overline{[\![\Gamma]\!]}}\ Tclose$$

$$\left[\!\left[\frac{\Gamma \text{ unr}}{\Gamma, x : \tau \ \vdash \ x \ : \ \tau}\right]\!\right]_z = \frac{\dfrac{}{\mathbf{fwd}\ x\ z \vdash_\eta x : \overline{[\![\tau]\!]}, z : [\![\tau]\!]; \overline{[\![\Gamma]\!]}}\ Tfwd}{\mathbf{use}\ x; \mathbf{fwd}\ x\ z \vdash_\eta x : \vee\overline{[\![\tau]\!]}, z : [\![\tau]\!]; \overline{[\![\Gamma]\!]}}\ Tuse$$

$$\left[\!\left[\frac{\Gamma \equiv \Gamma_1 \cdot \Gamma_2 \quad \Gamma_1 \ \vdash \ e_1 \ : \ \tau_1 \multimap \tau_2 \quad \Gamma_2 \ \vdash \ e_2 \ : \ \tau_1}{\Gamma \ \vdash \ e_1\ e_2 \ : \ \tau_2}\right]\!\right]_z =$$

$$\frac{[\![e_1]\!]_x \vdash_\eta \vee\overline{[\![\Gamma_1]\!]}, x : \vee\overline{[\![\tau_1]\!]} \parr \overline{[\![\tau_2]\!]}; \overline{[\![\Gamma]\!]} \qquad \dfrac{\dfrac{[\![e_2]\!]_y \vdash_\eta \vee\overline{[\![\Gamma_2]\!]}, y : [\![\tau_1]\!]; \overline{[\![\Gamma]\!]}}{\mathbf{affine}_{c,a}\ y; [\![e_2]\!]_y \vdash_\eta \vee\overline{[\![\Gamma_2]\!]}, y : \wedge[\![\tau_1]\!]; \overline{[\![\Gamma]\!]}}\ Taffine \quad \dfrac{}{\mathbf{fwd}\ x\ z \vdash_\eta x : \overline{[\![\tau_2]\!]}, z : [\![\tau_2]\!]; \overline{[\![\Gamma]\!]}}\ Tfwd}{\mathbf{send}\ x(y.\mathbf{affine}_{c,a}\ y; [\![e_2]\!]_y); \mathbf{fwd}\ x\ z \vdash_\eta \vee\overline{[\![\Gamma_2]\!]}, x : \wedge[\![\tau_1]\!] \otimes \overline{[\![\tau_2]\!]}, z : [\![\tau_2]\!]; \overline{[\![\Gamma]\!]}}\ Tsend}{\mathbf{cut}\ \{[\![e_1]\!]_x\ |x : \vee\overline{[\![\tau_1]\!]} \parr [\![\tau_2]\!]|\ \mathbf{send}\ x(y.\mathbf{affine}_{c,a}\ y; [\![e_2]\!]_y); \mathbf{fwd}\ x\ z\} \vdash_\eta \vee\overline{[\![\Gamma]\!]}, z : [\![\tau_2]\!]; \overline{[\![\Gamma]\!]}}\ Tcut$$

$$\left[\!\left[\frac{\Gamma, x : \tau_1 \ \vdash \ e \ : \ \tau_2}{\Gamma \ \vdash \ \lambda x.e \ : \ \tau_1 \multimap \tau_2}\right]\!\right]_z = \frac{[\![e]\!]_z \vdash_\eta \vee\overline{[\![\Gamma]\!]}, x : \vee\overline{[\![\tau_1]\!]}, z : [\![\tau_2]\!]; \overline{[\![\Gamma]\!]}}{\mathbf{recv}\ z(x); [\![e]\!]_z \vdash_\eta \vee\overline{[\![\Gamma]\!]}, z : \vee\overline{[\![\tau_1]\!]} \parr [\![\tau_2]\!]; \overline{[\![\Gamma]\!]}}\ Trecv$$

$$\left[\!\left[\frac{\Gamma \equiv \Gamma_1 \cdot \Gamma_2 \quad \Gamma_1 \ \vdash \ e_1 \ : \ \tau_1 \to \tau_2 \quad \Gamma_2 \ \vdash \ e_2 \ : \ \tau_1}{\Gamma \ \vdash \ e_1\ e_2 \ : \ \tau_2}\right]\!\right]_z =$$

$$\frac{[\![e_1]\!]_x \vdash_\eta x :!(\vee\overline{[\![\tau_1]\!]} \parr [\![\tau_2]\!]); \overline{[\![\Gamma]\!]} \quad Q \vdash_\eta \vee\overline{[\![\Gamma]\!]}, x :? \wedge [\![\tau_1]\!] \otimes \overline{[\![\tau_2]\!]}, z : [\![\tau_2]\!]; \overline{[\![\Gamma]\!]}}{\mathbf{cut}\ \{[\![e_1]\!]_x\ |x :!(\vee\overline{[\![\tau_1]\!]} \parr [\![\tau_2]\!])|\ ?x; \mathbf{call}\ x(x'); \mathbf{send}\ x'(y.\mathbf{affine}_{c,a}\ y; [\![e_2]\!]_y); \mathbf{fwd}\ x\ z\} \vdash_\eta \vee\overline{[\![\Gamma]\!]}, z : [\![\tau_2]\!]; \overline{[\![\Gamma]\!]}}\ Tcut$$

with

$$\dfrac{\dfrac{\dfrac{[\![e_2]\!]_y \vdash_\eta \vee\overline{[\![\Gamma]\!]}, y : [\![\tau_1]\!]; \overline{[\![\Gamma]\!]}, x : \wedge[\![\tau_1]\!] \otimes \overline{[\![\tau_2]\!]}}{\mathbf{affine}_{c,a}\ y; [\![e_2]\!]_y \vdash_\eta \vee\overline{[\![\Gamma]\!]}, y : \wedge[\![\tau_1]\!]; \overline{[\![\Gamma]\!]}, x : \wedge[\![\tau_1]\!] \otimes \overline{[\![\tau_2]\!]}}\ Taffine \quad \dfrac{}{\mathbf{fwd}\ x'\ z \vdash_\eta x' : \overline{[\![\tau]\!]}, z : [\![\tau_2]\!]; \overline{[\![\Gamma]\!]}, x : \wedge[\![\tau_1]\!] \otimes \overline{[\![\tau_2]\!]}}\ Tfwd}{\mathbf{send}\ x'(y.\mathbf{affine}_{c,a}\ y; [\![e_2]\!]_y); \mathbf{fwd}\ x'\ z \vdash_\eta \vee\overline{[\![\Gamma]\!]}, x : \wedge[\![\tau_1]\!] \otimes \overline{[\![\tau_2]\!]}, z : [\![\tau_2]\!]; \overline{[\![\Gamma]\!]}, x : \wedge[\![\tau_1]\!] \otimes \overline{[\![\tau_2]\!]}}\ Tsend}{\mathbf{call}\ x(x'); \mathbf{send}\ x'(y.\mathbf{affine}_{c,a}\ y; [\![e_2]\!]_y); \mathbf{fwd}\ x\ z \vdash_\eta \vee\overline{[\![\Gamma]\!]}, z : [\![\tau_2]\!]; \overline{[\![\Gamma]\!]}, x : \wedge[\![\tau_1]\!] \otimes \overline{[\![\tau_2]\!]}}\ Tcall}{Q = ?x; \mathbf{call}\ x(x'); \mathbf{send}\ x'(y.\mathbf{affine}_{c,a}\ y; [\![e_2]\!]_y); \mathbf{fwd}\ x\ z \vdash_\eta \vee\overline{[\![\Gamma]\!]}, x :? \wedge [\![\tau_1]\!] \otimes \overline{[\![\tau_2]\!]}, z : [\![\tau_2]\!]; \overline{[\![\Gamma]\!]}}\ T?$$

$$\left[\!\left[\frac{\Gamma \text{ unr} \quad \Gamma, x : \tau_1 \ \vdash \ e \ : \ \tau_2}{\Gamma \ \vdash \ \lambda x.e \ : \ \tau_1 \to \tau_2}\right]\!\right]_z = \frac{\dfrac{[\![e]\!]_z \vdash_\eta x : \vee\overline{[\![\tau_1]\!]}, y : [\![\tau_2]\!]; \overline{[\![\Gamma]\!]}}{\mathbf{recv}\ y(x); [\![e]\!]_y \vdash_\eta y : \vee\overline{[\![\tau_1]\!]} \parr [\![\tau_2]\!]; \overline{[\![\Gamma]\!]}}\ Trecv}{z(y); \mathbf{recv}\ y(x); [\![e]\!]_y \vdash_\eta z :!(\vee\overline{[\![\tau_1]\!]} \parr [\![\tau_2]\!]); \overline{[\![\Gamma]\!]}}\ T!$$

$$\left[\!\left[\frac{\Gamma \equiv \Gamma_1 \cdot \Gamma_2 \quad \Gamma_1 \ \vdash \ e_1 \ : \ \tau_1 \quad \Gamma_2 \ \vdash \ e_2 \ : \ \tau_2}{\Gamma \ \vdash \ (e_1, e_2) \ : \ \tau_1 \times \tau_2}\right]\!\right]_z =$$

$$\frac{\dfrac{[\![e_1]\!]_x \vdash_\eta \vee\overline{[\![\Gamma_1]\!]}, x : [\![\tau_1]\!]; \overline{[\![\Gamma]\!]}}{\mathbf{affine}_{c,a}\ x; [\![e_1]\!]_x \vdash_\eta \vee\overline{[\![\Gamma_1]\!]}, x : \wedge[\![\tau_1]\!]; \overline{[\![\Gamma]\!]}}\ Taffine \quad \dfrac{[\![e_2]\!]_z \vdash_\eta \vee\overline{[\![\Gamma_2]\!]}, z : [\![\tau_2]\!]; \overline{[\![\Gamma]\!]}}{\mathbf{affine}_{c,a}\ z; [\![e_2]\!]_z \vdash_\eta \vee\overline{[\![\Gamma_2]\!]}, z : \wedge[\![\tau_2]\!]; \overline{[\![\Gamma]\!]}}\ Taffine}{\mathbf{send}\ z(x.[\![e_1]\!]_x); [\![e_2]\!]_z \vdash_\eta \vee\overline{[\![\Gamma]\!]}, z : \wedge[\![\tau_1]\!] \otimes \wedge[\![\tau_2]\!]; \overline{[\![\Gamma]\!]}}\ Tsend$$

$$\left[\!\left[\frac{\Gamma \equiv \Gamma_1 \cdot \Gamma_2 \quad \Gamma_1 \ \vdash \ e_1 \ : \ \tau_1 \times \tau_2 \quad \Gamma_2, x\tau_1, y : \tau_2 \ \vdash \ e_2 \ : \ \tau_3}{\Gamma \ \vdash \ \mathbf{let}\ x, y = e_1\ \mathbf{in}\ e_2 \ : \ \tau_1 \times \tau_2}\right]\!\right]_z =$$

$$\frac{[\![e_1]\!]_x \vdash_\eta \vee\overline{[\![\Gamma_1]\!]}, x : \wedge[\![\tau_1]\!] \otimes \wedge[\![\tau_2]\!]; \overline{[\![\Gamma]\!]} \qquad \dfrac{[\![e_2]\!]_z \vdash_\eta \vee\overline{[\![\Gamma_2]\!]}, x : \vee\overline{[\![\tau_1]\!]}, y : \vee\overline{[\![\tau_2]\!]}, z : [\![\tau_2]\!]; \overline{[\![\Gamma]\!]}}{\mathbf{recv}\ x(y); [\![e_2]\!]_z \vdash_\eta \vee\overline{[\![\Gamma_2]\!]}, x : \vee\overline{[\![\tau_1]\!]} \parr \ \vee\overline{[\![\tau_2]\!]}, z : [\![\tau_2]\!]; \overline{[\![\Gamma]\!]}}\ Trecv}{\mathbf{cut}\ \{[\![e_1]\!]_x\ |x : \wedge[\![\tau_1]\!] \otimes \wedge[\![\tau_2]\!]|\ \mathbf{recv}\ x(y); [\![e_2]\!]_z\} \vdash_\eta \vee\overline{[\![\Gamma]\!]}, z : [\![\tau_3]\!]; \overline{[\![\Gamma]\!]}}\ Tcut$$

Figure 3: Encoding of type rules of base calculus, part 1

$$\left[\!\left[\frac{\Gamma \equiv \Gamma_1 \cdot \Gamma_2 \quad \Gamma_1 \vdash e : \tau_1 + \tau_2 \quad \Gamma_2, x_1 : \tau_1 \vdash e_1 : \tau' \quad \Gamma_2, x_2 : \tau_2 \vdash e_2 : \tau'}{\Gamma \vdash \mathbf{match}\ e\ \mathbf{with}\ \mathbf{in}_L(x_1) \Rightarrow e_1; \mathbf{in}_R(x_2) \Rightarrow e_2;\ \mathbf{end} : \tau'} \right]\!\right]_z =$$

$$\frac{[\![e]\!]_x \vdash_\eta \vee\overline{[\![\Gamma_1]\!]}, x : \wedge[\![\tau_1]\!] \oplus \wedge[\![\tau_2]\!]; \overline{[\![\Gamma]\!]} \qquad \frac{[\![e_1]\!]_z \vdash_\eta \vee\overline{[\![\Gamma_2]\!]}, x : \vee\overline{[\![\tau_1]\!]}, z : [\![\tau']\!]; \overline{[\![\Gamma]\!]} \quad [\![e_2]\!]_z \vdash_\eta \vee\overline{[\![\Gamma_2]\!]}, x : \vee\overline{[\![\tau_2]\!]}, z : [\![\tau']\!]; \overline{[\![\Gamma]\!]}}{\mathbf{case}\ x\ \{|\mathrm{inl} : [\![e_1]\!]_z | \mathrm{inr} : [\![e_2]\!]_z\} \vdash_\eta \vee\overline{[\![\Gamma_2]\!]}, x : \vee\overline{[\![\tau_1]\!]}\ \&\ \vee\overline{[\![\tau_2]\!]}, z : [\![\tau']\!]; \overline{[\![\Gamma]\!]}}\ T\&}{\mathbf{cut}\ \{[\![e]\!]_x\ |x : \wedge[\![\tau_1]\!] \oplus \wedge[\![\tau_2]\!]|\ \mathbf{case}\ x\ \{|\mathrm{inl} : [\![e_1]\!]_z | \mathrm{inr} : [\![e_2]\!]_z\}\} \vdash_\eta \vee\overline{[\![\Gamma]\!]}, z : [\![\tau']\!]; \overline{[\![\Gamma]\!]}}\ Tcut$$

$$\left[\!\left[\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathbf{in}_L(e) : \tau_1 + \tau_2} \right]\!\right]_z =$$

$$\frac{\dfrac{[\![e]\!]_z \vdash_\eta \vee\overline{[\![\Gamma]\!]}, z : [\![\tau_1]\!]; \overline{[\![\Gamma]\!]}}{\mathbf{affine}_{c,a}\ z; [\![e]\!]_z \vdash_\eta \vee\overline{[\![\Gamma]\!]}, z : \wedge[\![\tau_1]\!]; \overline{[\![\Gamma]\!]}}\ Taffine}{z.\mathbf{inl}; \mathbf{affine}_{c,a}\ z; [\![e]\!]_z \vdash_\eta \vee\overline{[\![\Gamma]\!]}, z : \wedge[\![\tau_1]\!] \oplus \wedge[\![\tau_2]\!]; ctd[\![\Gamma]\!]}\ T\oplus_L$$

$$\left[\!\left[\frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathbf{in}_R(e) : \tau_1 + \tau_2} \right]\!\right]_z =$$

$$\frac{\dfrac{[\![e]\!]_z \vdash_\eta \vee\overline{[\![\Gamma]\!]}, z : [\![\tau_2]\!]; \overline{[\![\Gamma]\!]}}{\mathbf{affine}_{c,a}\ z; [\![e]\!]_z \vdash_\eta \vee\overline{[\![\Gamma]\!]}, z : \wedge[\![\tau_2]\!]; \overline{[\![\Gamma]\!]}}\ Taffine}{z.\mathbf{inr}; \mathbf{affine}_{c,a}\ z; [\![e]\!]_z \vdash_\eta \vee\overline{[\![\Gamma]\!]}, z : \wedge[\![\tau_1]\!] \oplus \wedge[\![\tau_2]\!]; \overline{[\![\Gamma]\!]}}\ T\oplus_L$$

Figure 4: Encoding of type rules of base calculus, part 2