Summary

This report introduces a typed translation from the language basic typed Futhark (BtF), to the process calculus typed extended π-calculus (TEπ). This report extends the work by Jensen et. al in which they introduce the language basic untyped Futhark (ButF), the process calculus extended π-calculus ($E\pi$) and the translation from ButF to Eπ.
We start by introducing ButF - a core language based on the functional array programming language Futhark which compiles to optimised graphical processing unit (GPU) code. What makes Futhark interesting is the addition of arrays in a functional programming language and array operations known as second-order array combinators (SOACs). The semantics of SOACs allow Futhark to rewrite expressions which helps with optimisation.
Next we introduce Eπ, which is a process calculus based on the applied π-calculus and extended with broadcasting capabilities and composite names to better handle the array structure for the translation of ButF to Eπ. Then we go over the original translation and give some examples which shows translations of ButF expressions.

To bring ButF closer to Futhark we extend it with a type system, incorporating types such integers, arrays, and tuples inspired by Futhark's simple type system and the simply typed λ-calculus. By introducing a type system we acquire static guarantees and ensure illogical expressions, such as a binary operations on two abstractions, is not allowed. For BtF we introduce a theorem for soundness of the type system which consists of two parts: preservation (types are preserved after a transition step) and progress (expressions are either values or can take a transition step). We then give a proof of the soundness theorem.
This is followed by introducing a type system for Eπ. To handle composite we introduce a location type and pre-channel type which restrict how we combine names through the type rules and by introducing a pre-channel type environment - a type environment to ensure we handle arrays and tuples in the translation correctly. This is followed by a soundness theorem consisting of two parts: subject reduction (types are preserved after a reduction) and type safety (if we are well-typed reduction errors can not occur). For type safety we introduce an *error* predicate and its reduction rules, and give the proof for soundness.

Lastly we give a typed translation from BtF to TEπ and show previous examples updated with the new translation. To prove the correctness of the translation we introduce theorems about type correctness and behavioral correctness.
Type correctness ensures that every well-typed BtF expression translates to a well-typed TEπ process, with the type carried by the output channel type mirroring the type of the expression. Behavioral correctness ensures the translation behaves similar to BtF on the observable output after an important reduction. For this we introduce an operational correspondence that relates BtF expressions and TEπ processes. Together, these results provide a type-preserving translation of BtF to TEπ.

We conclude the report and discuss future work which includes ideas for possible improvements in the semantics, extending BtF with a new construct from Futhark called `with`, and introducing sized types to BtF.

A Type Safe Translation of a Functional Array Language to a Process Calculus

cs-25-sv-10-01
Daniel Vang Kleist, and Loke Walsted

AALBORG UNIVERSITY
STUDENT REPORT

Computer Science
Aalborg University
http://cs.aau.dk

Title:
A Type Safe Translation of a Functional Array Language to a Process Calculus

Theme:
Semantics and type systems

Project Period:
Spring Semester 2025

Project Group:
cs-25-sv-10-01

Participants:
Daniel Vang Kleist
Loke Walsted

Supervisor:
Hans Hüttel

Date:
2025-06-06

Copies: 1

Page Numbers: 41

Abstract:
Using graphics processing units (GPUs) for computations bring a significant amount of computational power compared to the central processing unit (CPU) through the use parallelisation. The functional array programming language Futhark makes use of array operations known as second-order array combinators (SOACs) for compiling to optimised GPU code. In this report we extend an existing untyped core language, ButF (a subset of Futhark), and an extended π-calculus with broadcasting capabilities, E π, by Hüttel et. al, with simple type systems (called BtF and TEπ, respectively). The two type systems are then proven to be sound in regards to their respective semantics. The original translation from ButF to Eπ is then extended by incorporating the type systems which provides static guarantees about program behaviors. This results in a type-preserving data-parallel implementation of BtF in TEπ which is proven to be correct through proofs about the translation of the types and behaviour.

Contents

1 Introduction 1
1.1 Futhark - a Functional Array Programming Language 1
1.2 The π-calculus - a Process Calculus 1
1.3 Translations to the π-calculus 2
1.4 Structure of the Report 2

2 Preliminaries 3
2.1 Basic Untyped Futhark 3
2.2 Extended π-calculus 5
2.3 Translation of ButF to Eπ 8

3 A Typed Setting 14
3.1 Basic Typed Futhark 14
3.2 Soundness of BtF 16
3.3 Typed Eπ 19
3.4 Soundness of TEπ 22

4 Translation and Correctness 26
4.1 Translation of BtF to TEπ 26
4.2 Correctness of the Translation 32

5 Conclusion 39
5.1 Results 39
5.2 Future Work 40

Bibliography 41

Appendix

A Appendix for Preliminaries I
A.1 ButF Definitions I
A.2 ButF Semantics II
A.3 Eπ Definitions III

B Proofs About the Type System for BtF VII
B.1 Proof of Lemma 3.1 VII
B.2 Proof of Lemma 3.2 VII
B.3 Proof of Lemma 3.3 VIII
B.4 Proof of Theorem 3.1 - preservation IX
B.5 Proof of Theorem 3.1 - progress XI

C The *error* predicate of TEπ XIV

D Proofs About the Type System for TEπ XV
D.1 Proof of Lemma 3.4 XV
D.2 Proof of Lemma 3.5 XV
D.3 Proof of Lemma 3.6 XVI
D.4 Proof of Lemma 3.7 XVII
D.5 Proof of Lemma 3.8 XVIII
D.6 Proof of Lemma 3.9 XIX
D.7 Proof of Lemma 3.10 XX

D.8 Proof of Theorem 3.2 - subject reduction XXII
D.9 Proof of Theorem 3.2 - type safety XXIII

E Proofs for the Translation of BtF to TEπ XXVI
E.1 Proof of Lemma 4.1 XXVI
E.2 Proof of Lemma 4.2 XXVI
E.3 Proof of Lemma 4.3 XXXII
E.4 Proof of Lemma 4.4 XXXII
E.5 Proof of Lemma 4.5 XXXIII
E.6 Proof of Lemma 4.6 XXXIV
E.7 Proof of Theorem 4.1 XXXV
E.8 Proof of Theorem 4.2 XXXIX

1 Introduction

A graphics processing unit (GPU) is a central part of most modern computers, mostly used for image rendering and displaying it to a monitor. This is, of course, not the only kind of computation GPUs can be used for, and through the introduction of programming interfaces such as CUDA [3] and later OpenCL [4], writing GPU code without knowing shader languages was made possible [5]. One key advantage of using a GPU for computations instead of the Central Processing Unit (CPU) is that the computational power of the GPU is much larger compared to that of the CPU. The reason for this is the utilisation of parallelisation in GPUs [5].

1.1 Futhark - a Functional Array Programming Language

Futhark is a functional array programming language that generates GPU code by CUDA and OpenCL and is designed with the focus on compiling to optimised GPU code [6]. The language is an ongoing research project but can still be used for non-trivial programs. The foundation of the Futhark language is that of array manipulation using what is known as second-order array combinators (SOACs) [7]. This is based on the work by Bird, in which some theoretical groundwork for manipulating list/array functions is created [8]. This includes the following functions: `map`, `reduce`, `foldl`, `foldr` and `scan`.

The Futhark semantics of SOACs allow for rewrites of expressions, which helps with optimisation by not only transforming the schedule but also the space [7]. Futhark utilises a transformation of parallelism for optimisation by taking nested parallelism and reorganising it into SOACs nests (that being the outer levels correspond to `map` operators) [7]. Futhark is syntactically similar to other functional programming languages such as Haskell but focuses less on the expressivity and type systems.

1.2 The π-calculus - a Process Calculus

To help prove aspects of data-parallelism in Futhark, one might look towards process calculi, a tool useful for verifying and proving certain behaviours of systems. In the family of process calculi, the π-calculus is one which can describe concurrent parallel systems. First introduced by Milner et. al in [9], [10], the π-calculus is a process calculus in which communication happens by name passing.

Though the π-calculus is very expressive, extensions and sub-calculi exist which allow for shorter and less cumbersome notation for writing the same process. Take for example the polyadic π-calculus [11] where multiple names can be sent and received in communication, or the Higher-Order π-calculus (HOπ) [12] where processes can be communicated. Both have been shown to be encodable in the monadic π-calculus. The polyadic π-calculus by Milner in [11] and HOπ by Sangiorgi in [12]. There also exist variations which cannot be expressed in the π-calculus. One such is $b\pi$-calculus (a calculus with broadcast derived from the π-calculus), which Ene and Muntean show that no uniform encoding of $b\pi$-calculus into the π-calculus exists [13].

One of the approaches for proving certain properties of a programming language or process calculus is that of encoding the original to another language or process calculus such as the π-calculus. Translating Futhark to the π-calculus would give a data-parallel implementation for free. This would also make it possible to analyse the complexity of data-parallelism in Futhark.

1.3 Translations to the π-calculus

One of the most well-known encodings is Milner's encodings of the call-by-value λ-calculus and the lazy λ-calculus to the π-calculus [14]. In [15], Sangiorgi studies the relationship between the encodings of the call-by-value and call-by-name λ-calculus to π-calculus. The approach taken by Sangiorgi is different than Milner's. In [15] the encoding is obtained by transforming the λ-calculus into a continuation passing style which is then translated into HOπ and lastly into the π-calculus.

In [16], Honda et. al introduce a type-preserving translation of the $\lambda\mu$-calculus (a variation of the λ-calculus with continuation variables) to a subset of the asynchronous π-calculus. Another type-preserving encoding can be found in [17], where Amadio et. al introduce a translation from a concurrent λ-calculus (a variation of the call-by-value λ-calculus extended with parallel composition, restriction, output and input) to the π-calculus. The concurrent λ-calculus is an attempt at capturing the nature of a concurrent functional programming language.

Work regarding encodings of languages to the π-calculus is also prevalent and used for showing or proving properties. Walker gives a translation of two different object-oriented languages, respectively, to illustrate the expressiveness of the π-calculus [18]. A translation of the functional programming language Erlang can be found in [19]. By translating a subset of Erlang known as core Erlang to the asynchronous π-calculus, model checking techniques could possibly be used for verifying correctness properties of communication systems implemented in Erlang [19].

In the work by Hüttel et. al, a subset of the Futhark language (called ButF) is translated to an extended π-calculus (called $E\pi$) [2]. This translation is then used for an analysis of the complexities of expressions and compared to that of Futhark. As ButF is untyped, this allows for writing expressions that logically make no sense; it is for example allowed to use a binary operation on two abstractions. This brings up the question of how one could design a type system for ButF to bring the language closer to that of Futhark. That would also restrict the language such that these illogical expressions cannot be valid. The type system of Futhark has three aspects; simple types, unique types, and sized types. We will in this report introduce ButF and $E\pi$ and extend both with a type system, respectively. The type system we introduce for ButF will try to embody that of the simple types of Futhark as introduced in [20]. With the inclusion of a type system we gain static guarantees in regards to the behaviour of a program which a correct translation would preserve. For this we need an updated translation and proofs for the correctness of the translation.

1.4 Structure of the Report

The remainder of the report is structured as follows: in Chapter 2 we introduce the ButF language, Eπ, and the translation from ButF to Eπ by Hüttel et. al from [2]. In Chapter 3 we extend ButF with a type system (we call this BtF) and prove the soundness of the type system. This is followed by an extension with a type system to Eπ (we call this TEπ), where we also prove the soundness of this type system. In Chapter 4 we introduce an updated translation from BtF to TEπ and define and prove the correctness of the translation. Lastly, in Chapter 5 we conclude the report and discuss future work.

2 Preliminaries

We will in this chapter give an introduction to language ButF, the process calculus Eπ and the original translation from ButF to Eπ. We will start by introducing ButF as seen in [2] with a revised semantics. This is followed by an introduction of $E\pi$ and the translation from ButF to $E\pi$ as seen in [2].

2.1 Basic Untyped Futhark

In [2], Hüttel et. al introduces a reduced syntax for Futhark without types called Basic Untyped Futhark (ButF). ButF is made with the focus on array computations of Futhark and while ButF has removed most of the array operations, the core concept of Futhark still remains. This makes ButF very similar to a λ-calculus extended with arrays, tuples, and binary operations. Though Futhark has more array operations than ButF, Hüttel et. al have chosen the operations `map`, `size` and `iota` since they can be used to define other common array operations [2].

2.1.1 Syntax for ButF

$$
\begin{array}{lll}
e ::= b & & b ::= n \\
\quad x & & \quad \texttt{map} \\
\quad [e_1, ..., e_n] & & \quad \texttt{iota} \\
\quad e_1[e_2] & & \quad \texttt{size} \\
\quad \lambda x.e_1 & & \quad \odot \\
\quad e_1 e_2 & & \\
\quad (e_1, ..., e_n) & & \\
\quad \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 & &
\end{array}
$$

Figure 2.1: ButF syntax [2]

Expressions in ButF is built with the following constructs:

- b which are constants and can either be an integer constant (n), an array operation or an arithmetic operation. Array operations are either `map`, `iota` or `size`.
  - `map` takes a tuple containing a function and an array, and returns an array where the function has been applied to each element.
  - `size` takes an array and returns the length of the array.
  - `iota` takes an integer and returns an array with length equal to the integer given and with the value of each element equal to its index.
  - The arithmetic operator $\odot$ are the standard arithmetic operations such as $+, -, \cdot, /$ and $\%$.
- x denotes a variable, $[e_1, ..., e_n]$ an array and $(e_1, ..., e_n)$ a tuple.
- $e_1[e_2]$ (indexing) returns the value at the location in an array (e_1) based on the index (e_2).
- $\lambda x.e_1$ (abstraction) introduces a variable x in an expression.
- $e_1 e_2$ (application) applies expression e_1 to the expression e_2.
- The conditional expression takes e_1 and if it evaluates to true then we proceed as e_2 else we proceed as e_3.

2.1.2 Semantics for ButF

The operational semantics of ButF is a small-step semantics and is given as a reduction relation ($\rightarrow$) of the form $e \rightarrow e'$. The semantics we present is a revised semantics to the one introduced in [2]. The language ButF is a call-by-value language with values defined as follow.

Definition 2.1 (Value): We define a value as v in the set of all values $v \in \mathcal{V}$, where values are constants, function symbols, arrays and tuples that contain values only, and abstractions:

$$v := b \mid [v_1, ..., v_n] \mid (v_1, ..., v_n) \mid \lambda x.e$$

For the semantics of ButF we will show some of the rules specific to ButF's array operations. The full semantics of ButF can be found in Appendix A.2.

$$\text{(B-Index1)}\ \frac{e_1 \rightarrow e_1'}{e_1[e_2] \rightarrow e_1'[e_2]} \qquad \text{(B-Index2)}\ \frac{e_2 \rightarrow e_2'}{e_1[e_2] \rightarrow e_1[e_2']}$$

$$\text{(B-Index)}\ \frac{\forall i \in \{1, ..., n\}}{[v_1, ..., v_n][i] \rightarrow v_i} \qquad \text{(B-Iota)}\ \frac{}{\texttt{iota}\ n \rightarrow [0, 1, ..., n-1]}$$

$$\text{(B-Size)}\ \frac{}{\texttt{size}\ [v_1, ..., v_n] \rightarrow n}$$

$$\text{(B-Map)}\ \frac{}{\texttt{map}(\lambda x.e, [v_1, ..., v_n]) \rightarrow [e\{x \mapsto v_1\}, ..., e\{x \mapsto v_n\}]}$$

Figure 2.2: ButF semantics

For indexing we have three rules. (B-Index1) is used for evaluation the first sub-expression e_1 and (B-Index2) for evaluating sub-expression e_2. When both sub-expression has fully evaluated to an array of values and an indexing number, respectively, we can use (B-Index) to take the final step and get the value at the index.

For the array operations the semantic rules reflects the intuition quite well. (B-Map) takes a tuple containing a function and an array of evaluated expressions and applies the function to each element. When applying an abstraction we substitute with the value from the abstraction. Substitution in ButF is denoted as $\{x \mapsto y\}$ and is read as x is substituted with y. The definition for substitution in ButF can be found in Appendix A.1. (B-Iota) takes a number n and returns an array of sequentially increasing elements from 0 to $n - 1$. (B-Size) takes an array of evaluated expressions and returns a value which corresponds the number of elements in the array.

2.2 Extended π-calculus

The Extended π-calculus ($E\pi$) is a process calculus based on the applied π-calculus [21] extended with broadcast and composite names [1]. First introduced in [22], Jensen et. al looks at extending the pi-calculus to capture constructs and concepts of Futhark in the later introduced translation. Broadcast is added by Jensen et. al to prevent other communication from occurring when translating the `map` construct from ButF and composite names is introduced to capture how some constructs are related, such as elements in arrays.

2.2.1 Syntax for Eπ

$$
\begin{array}{lll}
P, Q, R, ... ::= \mathbf{0} & A ::= \overline{c}\langle\vec{T}\rangle & c ::= a \\
\quad P \mid Q & \quad c\big(\vec{x}\big) & \quad x \\
\quad !P & \quad \overline{c}{:}\langle\vec{T}\rangle & \quad x \cdot l \\
\quad A.P & & \quad a \cdot l \\
\quad \bullet\, P & & \\
\quad \nu a.P & & \\
\quad [T_1 \bowtie T_2]P, Q & &
\end{array}
$$

$$
\begin{array}{ll}
l ::= n & T ::= n \\
\quad x & \quad a \\
\quad \texttt{all} & \quad x \\
\quad \texttt{len} & \quad T \odot T \\
\quad \texttt{tup} &
\end{array}
$$

Figure 2.3: Eπ Syntax [2]

The syntax for the Eπ is split into different formation rules where $P, Q, R, ...$ are processes, A are actions, c are channels, and T are terms. We denote names with lowercase letters and use specific letters to differentiate between them: a being a channel name, x being a variable, n being a number and u when we do not differentiate between a and x. We assume variables and names are distinct. $\mathbf{0}$ is the inactive process that being a process which cannot reduce further. We will sometimes leave out the trailing inactive process and write P instead of $P.\mathbf{0}$. Parallel composition $P \mid Q$ consists of two processes in parallel and replication $!P$ constructs an unbounded number of the process P. The process $\bullet\, P$ is used to denote an important step which is used in the translation. Eπ also includes the match construct $[T_1 \bowtie T_2]P, Q$ where $\bowtie \in \{<, >, \leq, \geq, =, \neq\}$ and should be read as: when $T_1 \bowtie T_2$ evaluates to true it proceeds as P, otherwise Q. Restriction of a name to a process ($\nu a.P$) is limited to only channel names. There are three actions in Eπ:

- Send ($\overline{c}\langle\vec{T}\rangle$) that sends a list of terms $\vec{T}$ on the channel c.
- Receive ($c\big(\vec{x}\big)$) which receives a list of terms and binds it to $\vec{x}$ on the channel c.
- Broadcast ($\overline{c}{:}\langle\vec{T}\rangle$) that sends $\vec{T}$ to everyone that listens on the channel c simultaneously.

Restriction and input acts as a binder for names to processes. Channels in Eπ can be either channel names or variables a, and x respectively; additionally both names and variables can be composed with a label $a \cdot l$. Labels are used in [2] to select specific behaviour of channels e.g. arr $\cdot$ `len` (x) will get the length of an array where arr $\cdot$ `all` (x) will get all the elements of arr. The idea of composite names comes from [23], though one key difference is that the composition of names introduced by Carbone and Maffeis allow composite names of arity k where in Eπ it

is restricted to an arity of two. Terms can be either $n \in \mathbb{N}$, channel names, variables or binary operations where $\odot \in \{+, -, \cdot, /, \%\}$ [2].

2.2.2 Semantics for Eπ

The semantics of Eπ is in the style of a labelled reduction semantics similar to that of the reduction semantics for the π-calculus. The structural congruence relation is defined as usual [22].

(Rename)	$P \equiv P'$ by α-conversion	(Replicate)	$!P \equiv P \mid !P$
(Par-**0**)	$P \mid \mathbf{0} \equiv P$	(New-**0**)	$\nu a.\mathbf{0} \equiv \mathbf{0}$
(Par-A)	$P \mid (Q \mid R) \equiv (P \mid Q) \mid R$	(New-A)	$\nu a.\nu b.P \equiv \nu b.\nu a.P$
(Par-B)	$P \mid Q \equiv Q \mid P$	(New-B)	$P \mid \nu a.Q \equiv \nu a.(P \mid Q)$ when $a \notin \mathrm{fv}(P) \cup \mathrm{fn}(P)$

Figure 2.4: Structural congruence rules

The structural congruence rules given are quite common for most π-calculi and allow for rewrites of processes. (Par-**0**), (Par-A) and (Par-B) allow for restructuring parallel composition. (New-**0**) show that a restriction on the inactive process is inconsequential and (New-A) that the order of restrictions can be swapped. (New-B) show that the scope of a restriction can be moved to encompass a parallel process or the other way around, remove said process from the restriction.

$$\text{(E-Com)}\ \frac{}{\overline{c}\langle\vec{T}\rangle.P | c\big(\vec{x}\big).Q \xrightarrow{\tau} P \mid Q\left\{{}^{\vec{T}}/_{\vec{x}}\right\}}$$

$$\text{(E-Broad)}\ \frac{}{\overline{c}{:}\langle\vec{T}\rangle.Q \mid c\big(\vec{x}_1\big).P_1 | ... | c\big(\vec{x}_n\big).P_n \xrightarrow{:c} Q \mid P_1\left\{{}^{\vec{T}}/_{\vec{x}_1}\right\} \mid ... \mid P_n\left\{{}^{\vec{T}}/_{\vec{x}_n}\right\}}$$

$$\text{(E-Par1)}\ \frac{P \xrightarrow{\tau} P'}{P \mid Q \xrightarrow{\tau} P' \mid Q} \qquad \text{(E-Par2)}\ \frac{c \notin Q \quad P \xrightarrow{:c} P'}{P \mid Q \xrightarrow{:c} P' \mid Q}$$

$$\text{(E-Res1)}\ \frac{P \xrightarrow{q} P' \quad q \neq: a}{\nu a.P \xrightarrow{q} \nu a.P'} \qquad \text{(E-Res2)}\ \frac{P \xrightarrow{:c} P' \quad a \in c}{\nu a.P \xrightarrow{\tau} \nu a.P'}$$

$$\text{(E-Then)}\ \frac{}{[T_1 \bowtie T_2]P, Q \xrightarrow{\tau} P}\ \text{if } T_1 \bowtie T_2 \qquad \text{(E-Else)}\ \frac{}{[T_1 \bowtie T_2]P, Q \xrightarrow{\tau} Q}\ \text{if } T_1 \not\bowtie T_2$$

$$\text{(E-Struct)}\ \frac{P \equiv Q \qquad Q \xrightarrow{q} Q' \quad P' \equiv Q'}{P \xrightarrow{q} P'}$$

Figure 2.5: Labelled reduction rules for Eπ

The semantics for Eπ can be seen in Figure 2.5. It is similar to the one for the monadic π-calculus introduced by Milner in [14], however to handle broadcast a labelled reduction semantic is introduced to ensure that broadcast is handled before the process can continue. In the reduction rules for communication and broadcast, when communication occur we substitute the names $\vec{x}$ with the received value $\vec{T}$ denoted as $\left\{{}^{\vec{T}}/_{\vec{x}}\right\}$. The substitution rules for Eπ can be found in Appendix A.3. For the labelled semantic three arrow types are introduced: $\xrightarrow{\tau}$, $\xrightarrow{:c}$, and $\xrightarrow{q}$. The reduction $\xrightarrow{\tau}$ is the regular reduction as seen in the π-calculus. The $\xrightarrow{:c}$ reduction is the broadcast reduction which ensures that all parallel receivers on a broadcast channel receives the broadcasted value. Lastly, $\xrightarrow{q}$ can be either a $\xrightarrow{\tau}$ or $\xrightarrow{:c}$ reduction [22].

$$\text{(Admn)}\ \frac{P \xrightarrow{\tau} P'}{P \xrightarrow{\circ} P'} \qquad \text{(NonAdm)}\ \frac{P \to P'}{\bullet\, P \xrightarrow{\bullet} P'} \qquad \text{(Both)}\ \frac{P \xrightarrow{s} P' \quad s \in \{\circ, \bullet\}}{P \to P'}$$

Figure 2.6: labelled semantics for important and administrative reductions as seen in [2]

In addition to the labelled reduction semantics, a labelled semantics for administrative and important reductions is given in [2]. These two types of reductions are used in the translation from ButF to Eπ and will be important when we define the correctness of our typed translation in Chapter 4.

2.3 Translation of ButF to Eπ

We will now introduce the translation from ButF to Eπ by Hüttel et. al, [2]. This will be used to give a general understanding of the original translation and will be used when we compare the new translation from BtF to TEπ introduced in Chapter 4.1.

For a translation of a ButF expression e to the corresponding Eπ process, the notation $[\![e]\!]$ will be used for the translated Eπ process. The notation $[\![e]\!]_o$ is used when specifying an output channel (that being the channel o) as parameter for a translated process that is used such that we can communicate with that process, as seen in [2]. This approach originates from the work by Milner where two translations to the π-calculus is shown: one of the lazy λ-calculus and one of the call-by-value λ-calculus [14].

2.3.1 Translation

In the translation we will use certain lower-case letters for names to illustrate their function such as h being a channel name whose purpose is being a function, array or tuple handle and v being a name signifying a value; that being the name received from a process that has already finished evaluating. In the translation the notation $\bullet$ marks important reductions in Eπ that matches a transition in ButF [2].

Translation of Expressions

$$[\![x]\!]_o = \overline{o}\langle x\rangle$$

$$[\![n]\!]_o = \overline{o}\langle n\rangle$$

$$[\![\lambda x.e]\!]_o = \nu h.(\overline{o}\langle h\rangle \mid !h(x,r).[\![e]\!]_r)$$

$$[\![e_1 e_2]\!]_o = \nu o_1.\nu o_2.\Big([\![e_1]\!]_{o_1} \mid [\![e_2]\!]_{o_2} \mid o_1(h).o_2(v).\bullet\, \overline{h}\langle v, o\rangle\Big)$$

$$[\![e_1[e_2]]\!]_o = \nu o_1.\nu o_2.\Big([\![e_1]\!]_{o_1} \mid [\![e_2]\!]_{o_2} \mid o_1(h).o_2(i).\bullet\, [i \geq 0]h \cdot i(i,v).\overline{o}\langle v\rangle, \mathbf{0}\Big)$$

$$[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]_o = \nu o_1.\Big([\![e_1]\!]_{o_1} \mid o_1(n).\bullet\, [n \neq 0][\![e_2]\!]_o, [\![e_3]\!]_o\Big)$$

$$[\![e_1 \odot e_2]\!]_o = \nu o_1.\nu o_2.\Big([\![e_1]\!]_{o_1} \mid [\![e_2]\!]_{o_2} \mid o_1(v_1).o_2(v_2).\overline{o}\langle v_1 \odot v_2\rangle\Big)$$

Figure 2.7: Translation of basic ButF expressions

The translation of variables and numbers is the same as in [14] with the channel o being the link. In the translation of abstraction, the new name h is introduced, which is a handle to the process. As we cannot transmit functions, this works as a pointer to the function. A replicated process is receiving on h, waiting for the parameter (x) and a return channel (r) which is used in the translation of the expression $[\![e]\!]_r$.

In application, we have three processes in parallel, the translations of expressions $[\![e_1]\!]_{o_1}$ and $[\![e_2]\!]_{o_2}$ and a process waiting for an output from the two processes. Upon receiving h and v the process will output the value and return channel on h (the channel the value after evaluation will be output on). Recall that abstraction waits for an input on h and that connects abstraction and application.

Indexing is translated similarly to application. First, we evaluate the two sub-expressions and receive the array handle (the name h) from $[\![e_1]\!]_{o_1}$, and the index (the name i) from $[\![e_2]\!]o_2$.

Then we match the index and if its larger or equal to 0, we receive the value on the composite name $h \cdot i$ and outputs it, else we proceed as the inactive process $\mathbf{0}$.

For the translation of branching we make use of the match constructor in TEπ. We first evaluate the translation of sub-expression $[\![e_1]\!]_{o_1}$, which will then output the result on the channel o_1.

Upon receiving the name n (which specify a number as seen in Chapter 2.2.1) we match it with 0 and proceed as either $[\![e_2]\!]_o$ or $[\![e_3]\!]_o$. As branching can only proceed as either one of the two we can use the same output channel (o) for both of them.

In addition we have introduced the translation of binary operation which is missing in [2]. In the translation of binary operation we first evaluate the two sub-expressions and receive the values on their respective output channel. We then send the value on o with the corresponding TEπ binary operation.

Translation of Arrays and Tuples

To define the translation of arrays we need a process *Cell* as introduced in [2].

$$Cell(h, i, v) = !h \cdot \texttt{all}\ (r).\overline{r}\langle i, v\rangle \mid !\overline{h \cdot i}\langle i, v\rangle \tag{1}$$

The *Cell* process is defined with a handle (h), an index (i) and a value (v). The reason for the handle is that *Cell* is defined using the same approach as translation of functions in the π -calculus - the handle is pointer to the cell. *Cell* has two composed names: the first name ($h \cdot$ `all`) is waiting for a request for all the values in the array. The second composed name ($h \cdot i$) is waiting for a request for a specific element in the array.

$$[\![[e_1, ..., e_n]]\!]_o = \nu o_1 ..... \nu o_n . \nu h . \left(\prod_{i=1}^{n} [\![e_i]\!]_{o_i} \mid o_1(v_1) ..... o_n(v_n). \left(\prod_{i=1}^{n} Cell(h, i-1, v_i) \mid \overline{h \cdot \texttt{len}}\langle n\rangle \mid \overline{o}\langle h\rangle \right) \right)$$

$$[\![(e_1, ..., e_n)]\!]_o = \nu o_1 ..... \nu o_n . \left(\prod_{i=1}^{n} [\![e_i]\!]_{o_i} \mid o_1(v_1) ..... o_n(v_n) . \nu h . \left(!\overline{h \cdot \texttt{tup}}\langle v_1, ... v_n\rangle \mid \overline{o}\langle h\rangle \right) \right)$$

Figure 2.8: Translation of arrays and tuples

Tuples and arrays are translated similarly. For array we translate each sub-expression and receive the evaluated values on each of their output channel. When we have received the values we can then create the array structure using the *Cell* process for each of the values we have received. The sub-process $\overline{h \cdot \texttt{len}}\langle n\rangle$ outputs the length of the array which will be used in the translation of the array operation `size`.

For the translation of tuples, we first we translate each sub-expression and receive the evaluated values over each of their output channel as in arrays. The output channel $h \cdot$ `tup` is a restriction for communication such that a tuple can only be used in place expecting a tuple.

Translation of Array Operations

$$[\![\texttt{size}\ e_1]\!]_o = \nu o_1.\Big([\![e_1]\!]_{o_1} \mid o_1(h).h \cdot \texttt{len}\ (n).\bullet\, \overline{o}\langle n\rangle\Big)$$

$$[\![\texttt{iota}\ e_1]\!]_o = \nu o_1.\nu r.\nu h.\Big([\![e_1]\!]_{o_1} \mid o_1(n).\ Repeat(n,r,d) \mid$$

$$!r(i,v).\ Cell(h,i,v) \mid \bullet\, d().\big(\overline{h.\ \texttt{len}}\langle n\rangle \mid \overline{o}\langle h\rangle\big)\Big)$$

$$[\![\texttt{map}\ e_1]\!]_o = \nu o_1.\nu h_1.\Big([\![e_1]\!]_{o_1} \mid o_1(x).x \cdot \texttt{tup}\ (f,h).h \cdot \texttt{len}\ (n).$$

$$\nu v_1.\overline{h \cdot \texttt{all}}{:}\langle v_1\rangle.\nu r.\nu d.\Big(Repeat(n,r,d) \mid$$

$$!v_1(i,v).\nu r_1.\overline{f}\langle v,r_1\rangle.r_1(v_2).r(\_,\_).\ Cell(h_1,i,v_2) \mid$$

$$\nu o_2.\overline{f}\langle 0,o_2\rangle.\bullet\, d().\overline{o}\langle h_1\rangle \mid !\overline{h_1 \cdot \texttt{len}}\langle n\rangle\Big)\Big)$$

Figure 2.9: Translation of array operations

One important thing to note of the translation of array operations is our presentation have some important reductions in the translation of `size` and `iota`, which cannot be seen in [2].

The translation of `size` is quite simple. First we translate the expression $[\![e_1]\!]$, which is an array. On its output channel we receive its handle which we can use to get the size of the array. Remember that the translation of arrays outputs the length over the channel $h \cdot \texttt{len}$. Here we receive the length of the array and will output on the return channel o.

For the translation of `map` and `iota` we introduce an auxiliary process called *Repeat* as seen in [2].

$$Repeat(s,r,d) = \nu c.(!c(n).\big([n \geq 0](\overline{r}\langle n-1, n-1\rangle \mid \overline{c}\langle n-1\rangle), \overline{d}\langle\rangle\big) \mid \overline{c}\langle s\rangle) \tag{2}$$

Repeat is, as the name suggest, a process that will repeatedly communicate with itself a number of times. This is similar to a restricted replication were instead of an unbounded number iteration the process is limited to a set number of iterations. Before a countdown can begin, an output of the initial value that will counted down from (that being s), is sent on c. This will trigger the next process that will do the actual repeating - this is seen by the replicated input channel c that upon receiving a number n does a match with 0 and will proceed with either:

1. Match succeeds: Output $n-1$ on the return channel r in parallel with outputting the same on its own channel c.
2. Match fails: Output an empty message on channel d to signal we are finished.

`iota` uses repeat with the number (n) we receive from evaluating the translated process $[\![e_1]\!]$. In parallel with *Repeat* we receive the index and the value (this being the same number) and then construct the array cell. When *Repeat* has finished we can send the array handle (h) on the output channel (o).

For the translation of `map` we first need the evaluation of the translation of $[\![e_1]\!]$ as that will return the arguments (x) on o_1. On the $x \cdot \texttt{tup}$ channel we will receive the function we will apply (f) and the array handle (h). With the array handle we get the length of the array which will be used later. We then do a broadcast to all array element of a certain name v_1, which will have all the values from the array. We use the *Repeat* process with the array length we

received earlier. In parallel we receive the index and value on v_1 followed by sending the value and return channel over the function channel (f). On this return channel we will receive the new value. We can then update the cell array, when have received on the *Repeat* return channel (r). The second last part is a check to ensure we are working with a function by sending a 0 over the function channel. We cannot proceed before *Repeat* has finished which we will know when we receive on channel d. Finally we can send the new array handle.

2.3.2 Examples of Translations

We will now give three examples of a translation from ButF to Eπ. The first is an example of indexing, the second of abstraction and the third of the map function. Different parts of the translations have been marked to help with readability.

Example of Indexing

This first example is the translation of the expression $[2,3][1]$, indexing of an array consisting of two numbers. When translating we will have the following form for indexing $[\![[2,3][1]]\!]_o$ expanding to Figure 2.10.

$$
\begin{aligned}
[\![[2,3][1]]\!]_o = \nu o_{\text{arr}}.(\nu o_1.(\nu o_2.(\nu o_3.(\nu h_1.(&\overbrace{\overline{o_2}\langle 2\rangle}^{[\![2]\!]_{o_2}} \mid \overbrace{\overline{o_3}\langle 3\rangle}^{[\![3]\!]_{o_3}} \mid \overbrace{o_2(v_1).o_3(v_2)}^{\text{receiver}} \\
&\mid \overbrace{!(h_1 \cdot \texttt{all}\ (r_0).\overline{r_0}\langle 0, v_1\rangle) \mid !(\overline{h_1 \cdot 0}\langle 0, v_1\rangle)}^{\text{cell\_0}} \\
&\mid \overbrace{!(h_1 \cdot \texttt{all}\ (r_1).\overline{r_1}\langle 1, v_2\rangle) \mid !(\overline{h_1 \cdot 1}\langle 1, v_2\rangle)}^{\text{cell\_1}} \mid \\
&\overbrace{!(\overline{h_1 \cdot \texttt{len}}\langle 2\rangle)}^{\text{len}} \mid \overbrace{\overline{o_{\text{arr}}}\langle h_1\rangle}^{\text{handle}}\ )))\ \mid \overbrace{\overline{o_1}\langle 1\rangle}^{\text{index}} \\
&\mid \overbrace{o_{\text{arr}}(h_0).o_1(i).[i \geq 0](h_0 \cdot i(i, v).\overline{o}\langle v\rangle), \mathbf{0})}^{\text{access}})
\end{aligned}
$$

Figure 2.10: Translation of the expression: [2,3][1]

If we take a look at the translation of indexing from Figure 2.7 we first need to translate the two sub-expressions: The first being the array ($[\![e_1]\!]$), the second being the number we want to index ($[\![e_2]\!]$).

For the first sub-expression (the translation of the array) we first need to translate each element in the array. This can be seen in the first two marks (2 and 3) where we output the number on their respective output channel. This matches the translation of numbers seen in Figure 2.7. The next step is receiving the values (marked with receiver), and then construct the array with the *Cell* process. The whole array consists of two cells (each marked as cell_0 and cell_1) constructed as seen in *Cell* process, the composite name with length of the array, and the handle of the array.

The second sub-expression is quite simple. It is just the output of the indexing number on the channel o_1 which has been marked with 1 above.

For the final part (marked as access), we receive the array handle on channel o_0 and the indexing number on channel o_1. This is followed by a match on the indexing number we received. If the match succeeds (which we know it will in this case) we can use the handle and index the get the value at that index, and output it on channel o. If the match fails we go to the **0** process.

Example of Abstraction

The second example is the translation of the expression $\lambda x.x+1$, an abstraction that takes a value x and adds 1 to it. When translating we will have the following form for abstraction $[\![\lambda x.x+1]\!]_o$ expanding to Figure 2.11

$$[\![\lambda x.x+1]\!]_o = \underbrace{\nu h_0.(\overline{o}\langle h_0\rangle \mid !(h_0(x,r_0).[\![x+1]\!]_r))}_{\text{lambda h\_0}}$$

$$[\![x+1]\!]_r = \underbrace{\nu o_0.(\nu o_1.(\overbrace{\overline{o_0}\langle x\rangle}^{x} \mid \overbrace{\overline{o_1}\langle 1\rangle}^{1} \mid \overbrace{\overbrace{o_0(v_0).o_1(v_1)}^{\text{receiver}}.\overbrace{\overline{r_0}\langle v_0+v_1\rangle}^{\text{binop}}}))}_{x+1}$$

Figure 2.11: Translation of $\lambda x.x+1$

If we take a look at the translation of a simple abstraction in Figure 2.11 we first need to create a function handle to receive the x and the return location r_0 and send it to the output channel o. Then we have to translate the sub-expression $[\![e]\!]_{r_0}$ which is a binary operation between a variable x and the number 1. The first part of the translation of the body $[\![e]\!]_{r_0}$ is translating the variable x and number 1 followed by receiving the values (marked with receiver), and then sending the result of the binary operation (marked binop) in r_0.

Example of Map

For our last example we have we have the expression `map` $(\lambda x.$ `size` $x, [[1,2],[3,4]])$, a mapping of the `size` operation to an array of arrays. The translation of `map` is quite long and therefore we have split it into smaller sub-translations.

First we have the translation of the two inner arrays.

$$[\![[1,2]]\!]_{o_B} = \nu o_1.\nu o_2.\nu h_A.(\overbrace{\overline{o_1}\langle 1\rangle \mid \overline{o_2}\langle 3\rangle}^{[\![1]\!]_{0_1} \text{ and } [\![2]\!]_{o_2}} \mid \overbrace{o_1(v_1).o_2(v_2)}^{\text{receiver}}.$$
$$\underbrace{(Cell(h_A,0,v_1) \mid Cell(h_A,1,v_2) \mid \overline{h_A \cdot \texttt{len}}\langle 2\rangle \mid \overline{o_A}\langle h_A\rangle)}_{\text{array: [1,2]}}$$

$$[\![[3,4]]\!]_{o_B} = \nu o_3.\nu o_4.\nu h_B.(\overbrace{\overline{o_3}\langle 3\rangle \mid \overline{o_4}\langle 4\rangle}^{[\![3]\!]_{0_3} \text{ and } [\![4]\!]_{o_4}} \mid \overbrace{o_3(v_3).o_4(v_4)}^{\text{receiver}}.$$
$$\underbrace{(Cell(h_B,0,v_3) \mid Cell(h_B,1,v_4) \mid \overline{h_B \cdot \texttt{len}}\langle 2\rangle \mid \overline{o_B}\langle h_B\rangle))}_{\text{array: [3,4]}}$$

The value located is each translated using the translation of numbers. When we have received the translated value (marked as receiver) we can then create the array using the *Cell* process:

$$Cell(h_B,0,v_3) = !h_B \cdot \texttt{all}\ (r).\overline{r}\langle 0,v_3\rangle \mid !\overline{h \cdot 0}\langle 0,v_3\rangle$$

Next we have the translation of the outer array. When we receive the array handles from the inner arrays we can construct the outer array using the *Cell* process with the handles as values on the index.

$$[\![[[1,2],[3,4]]]\!]_{o_{\text{arr}}} = \nu o_A.\nu o_B.\nu h.([\![[1,2]]\!]_{o_A} \mid [\![[3,4]]\!]_{o_B} \mid \overbrace{o_A(h_A).o_B(h_B)}^{\text{Receive inner array handles}} .$$
$$(\underbrace{Cell(h,0,h_A) \mid Cell(h,1,h_B) \mid \overline{h \cdot \texttt{len}}\langle 2\rangle \mid \overline{o_{\text{arr}}}\langle h\rangle)}_{\text{array: [[1,2],[3,4]]}})$$

We also have the translation of the abstraction using the size operation. Following the translation rules in Figure 2.7 and Figure 2.9 we get the following:

$$[\![\lambda x.\ \texttt{size}\ x]\!]_{o_F} = \overbrace{\nu h_F.(\overline{o_F}\langle h_F\rangle \mid\ !h_F(x,r).\underbrace{\nu o_x.(\overline{o_x}\langle x\rangle \mid o_x(h).h \cdot \texttt{len}\ (n).\bullet\, \overline{r}\langle n\rangle)}_{[\![\texttt{size}\ x]\!]_r})}^{\text{Abstraction}}$$

We can now create the tuple with the function, and the array we apply the function to. We insert the translation of the function and the array. When we receive the handles to both we can then output tuple handle on the $o_{\texttt{tup}}$ channel such that communication with the tuple can occur.

$$[\![(\lambda x.\ \texttt{size}\ x, [[1,2],[3,4]])]\!]o_{\texttt{tup}} = \nu o_F.\nu o_{\text{arr}}.(\overbrace{[\![\lambda x.\ \texttt{size}\ x]\!]_{o_F}}^{\text{Tuple elements}} \mid [\![[[1,2],[3,4]]]\!]_{o_{\text{arr}}}$$
$$\mid \underbrace{o_F(f).o_{\text{arr}}(h_{\text{in}})}_{\text{Function and array handle}} .\nu h_{\texttt{tup}}.(\underbrace{!\overline{h_{\texttt{tup}} \cdot \texttt{tup}}\langle f, h_{\text{in}}\rangle \mid \overline{o_{\texttt{tup}}}\langle h_{\texttt{tup}}\rangle}_{\text{Communication with the tuple}}))$$

Finally we have the translation of the full expression. First we have the translation of our tuple as seen above. When we have received all the handles from the tuple, we can then get the values from the input array. Then we can start the *Repeat* process to apply the function to each element. When we receive the value from the function we can the create the cell in the resulting array using the *Cell* process.

$$[\![\texttt{map}\ (\lambda x.\ \texttt{size}\ x, [[1,2],[3,4]])]\!]_o = \nu o_{\texttt{tup}}.\nu h_{\text{out}}.(\underbrace{[\![(\lambda x.\ \texttt{size}\ x, [[1,2],[3,4]])]\!]_{o_{\texttt{tup}}}}_{\text{Tuple}}$$
$$\mid \underbrace{o_{\texttt{tup}}(h_{\texttt{tup}}).h_{\texttt{tup}} \cdot \texttt{tup}\ (f, h_{\text{in}})}_{\text{Unpack tuple}}.\underbrace{h_{\text{in}} \cdot \texttt{len}\ (n)}_{\text{Input array size}}.\nu h_{\texttt{all}}.$$
$$\underbrace{\overline{h_{\text{in}} \cdot \texttt{all}}:\langle h_{\texttt{all}}\rangle}_{\text{Get input array elements}} .\nu r.\nu d.(Repeat(n,r,d) \mid$$
$$!h_{\texttt{all}}(i, v_{\text{sub}}).\nu r_1.\underbrace{\overline{f}\langle v_{\text{sub}}, r_1\rangle}_{\text{Function call}}.\underbrace{r_1(s_z)}_{\text{return value}} .r(\_,\_).$$
$$\underbrace{Cell(h_{\text{out}}, i, s_z)}_{\text{Fill resulting array}} \mid \underbrace{\bullet\, d()}_{\textit{Repeat}\text{ finished}} .\overline{o}\langle h_{\text{out}}\rangle \mid !\overline{h_{\text{out}} \cdot \texttt{len}}\langle n\rangle))$$

When we receive on d we know the *Repeat* process has finished, and so the resulting array after a `map` has been created which we can then output.

3 A Typed Setting

In this chapter, we introduce typed variations of ButF and Eπ, henceforth denoted as BtF and TEπ. We start by introducing a simple type system for ButF, an attempt in adapting the simple type system of Futhark introduced in [20] to BtF. Following this we provide a proof of soundness for BtF. Similarly, for Eπ we introduce a type system that handles composite names and prove the soundness of the type system.

3.1 Basic Typed Futhark

We extend ButF with a type system inspired by the original type system from [20] and the simply typed λ-calculus [24]. It should be noted that an original type system for ButF was introduced by Jensen et. al in [22] but the syntax and semantics for ButF has changed in [2].

3.1.1 Types for BtF

Type judgements are of the form: $\Gamma \vdash e : \tau$, where Γ is a type environment, e is a BtF expression and τ is a type. Type judgements should be read as: given an environment Γ, then the expression e has type τ.

Definition 3.1 (Type environment): An environment Γ is partial function from variables to types

$$\mathbf{Var} \rightharpoonup \boldsymbol{T}$$

We have chosen the primitive types of BtF based on the semantics from [2] and the Futhark documentation [25]. The primitive types introduced in [25] are signed and unsigned integers, floating-points, and boolean types. As of now, we have only chosen integers as in [2], but a future extension would be to have floating-point types as well as boolean types. In our case, booleans could have been used in the conditional expression but as seen in Figure 3.1 that is handled by evaluating the value of an integer where 0 is false and everything else is true.

Definition 3.2 (Types $\boldsymbol{T}$):

$$\begin{aligned} \tau ::= \ & \mathbf{Int} && \text{(Integer)} \\ \mid \ & [\tau] && \text{(Array)} \\ \mid \ & \left(\vec{\tau}\right) && \text{(Tuple)} \\ \mid \ & \tau_1 \to \tau_2 && \text{(Abstraction)} \end{aligned}$$

As mentioned earlier the only primitive type we have is **Int**. This simplifies the type system but still allow to show the core of Futhark. In addition to the primitive type **Int** we have two types for handling collections of elements: tuple types $\left(\vec{\tau}\right)$ and array types $[\tau]$. The tuple and array types can be nested meaning they can have an arbitrary depth. Lastly we have the abstraction type which can model higher order functions using tuples.

3.1.2 Syntax and Semantics of BtF

The syntax for BtF is the same as in ButF (Figure 2.1) however, the abstraction rule is changed to require a type annotation as it is the only binder construct in BtF.

$$
\begin{array}{ll}
e ::= b & b ::= n \\
\quad x & \quad \texttt{map} \\
\quad [e_1, ..., e_n] & \quad \texttt{iota} \\
\quad e_1[e_2] & \quad \texttt{size} \\
\quad \lambda(x : \tau).e_1 & \quad \odot \\
\quad e_1 e_2 & \\
\quad (e_1, ..., e_n) & \\
\quad \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 &
\end{array}
$$

Figure 3.1: BtF syntax

With the way the syntax of BtF is constructed we have three functions `map`, `iota` and `size` that needs a type to enforce correct usage. The way we solve this is by introducing an implicit type context (Definition 3.3). The implicit type context contains the types for the three array functions and is used to extend the type environment (Definition 3.1) as follows $\Gamma \cup \sum$.

Definition 3.3 (Implicit type context):

$$
\sum = \begin{Bmatrix}
\texttt{map} : (\tau_1 \to \tau_2, [\tau_1]) \to [\tau_2], \\
\texttt{iota} : \textbf{Int} \to [\textbf{Int}], \\
\texttt{size} : [\tau] \to \textbf{Int}
\end{Bmatrix}
$$

The semantics of BtF is the same as seen in Chapter 2.1.2 with a type annotation in abstraction.

3.1.3 Type rules

To adapt the simple type system of Futhark in BtF we took a look at the type rules introduced by Henriksen in [20]. In the cases where we were unable to create similar type rules due to the difference between BtF and Futhark we took inspiration from the simply typed λ-calculus. The type rules for BtF can be found in Figure 3.2.

(BT-If) is an interesting case to look at. We first look at the type of e_1 where it must be of type: **Int**, such that it complies with the semantics given in Figure 3.1. The most interesting part is that we enforce the types of the two branches to be the same. This agrees with what we encountered when we tested conditional branching in the Futhark language and the type rule found in [20].

Typing of (BT-Array) and (BT-Tuple) look very similar. For (BT-Array) we require the array to be homogeneous, that being each element of the array must have the same type. In comparison, tuples are heterogeneous, that being we allow each element to have a different type. If we restricted tuples in the same way as array our semantic rule for `map` would no longer hold.

In the (BT-Index) rule we need to type the two expressions. e_1 must be typed as an array type of τ_1 and e_2 as **Int**. The resulting type of index is τ_1 - the type of the elements in the array.

In the (BT-Abs) rule we need to type the abstraction body, the expression e where x has the type τ_1. The expression body has the type τ_2 if well-typed, giving the abstraction the resulting arrow type $\tau_1 \to \tau_2$.

In the (BT-App) rule, e_1 has to be typed as an arrow type $\tau_1 \to \tau_2$ otherwise e_1 is not an abstraction. We then require that e_2 is typed as τ_1 enforcing that the parameter has to have the correct type. The resulting type of application is the result type of the abstraction τ_2

$$\text{(BT-Int)}\ \frac{}{\Gamma \vdash n : \mathbf{Int}} \qquad \text{(BT-Var)}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

$$\text{(BT-Bin)}\ \frac{\Gamma \vdash e_1 : \mathbf{Int} \quad \Gamma \vdash e_2 : \mathbf{Int} \quad \Gamma \vdash \odot : \mathbf{Int} \to \mathbf{Int} \to \mathbf{Int}}{\Gamma \vdash e_1 \odot e_2 : \mathbf{Int}} \qquad \text{(BT-If)}\ \frac{\Gamma \vdash e_1 : \mathbf{Int} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}$$

$$\text{(BT-Array)}\ \frac{\forall i \in \{1, ..., n\} \Gamma \vdash e_i : \tau}{\Gamma \vdash [e_1, ..., e_n] : [\tau]} \qquad \text{(BT-Tuple)}\ \frac{\forall i \in \{1, ..., n\}\ \ \Gamma \vdash e_i : \tau_i}{\Gamma \vdash (e_1, ..., e_n) : (\tau_1, ..., \tau_n)}$$

$$\text{(BT-Index)}\ \frac{\Gamma \vdash e_1 : [\tau] \quad \Gamma \vdash e_2 : \mathbf{Int}}{\Gamma \vdash e_1[e_2] : \tau} \qquad \text{(BT-Abs)}\ \frac{\Gamma, (x : \tau_1) \vdash e : \tau_2}{\Gamma \vdash (\lambda x : \tau_1).e : \tau_1 \to \tau_2}$$

$$\text{(BT-App)}\ \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2}$$

Figure 3.2: Type rules of BtF

3.2 Soundness of BtF

With the type system introduced, one of the properties of the type system we want is that of soundness. For this we give the following theorem.

Theorem 3.1 (Soundness of BtF): Let e be a BtF expression then
1. (Preservation) if $\Gamma \vdash e : \tau$ and $e \to e$' then $\Gamma \vdash e' : \tau$
2. (Progress) if $\vdash e : \tau$ then $e = v \in \mathcal{V}$ or $\exists e'.e \to e'$

The soundness theorem consists of two parts: preservation and progress. Preservation states that if we are well-typed and can take a transition step then the resulting expression is also well-typed. Progress states that either the expression is a value or there exists an e' we can reach after taking a transition step.

3.2.1 BtF Lemmas

To help prove Theorem 3.1 we need formulate the following lemmas. Lemma 3.1 is necessary for connecting values and types. Lemma 3.2 states that we can add a new fresh variable to the type environment without conflicts. Lemma 3.3 states that substitution of variables are type preserving. The full proof of all the lemmas can be found in Appendix B.1 to Appendix B.3.

Lemma 3.1 (Inversion): If $\vdash v : \tau$ then
1. if $\tau = \textbf{Int}$ then v is a constant.
2. if $\tau = \tau_1 \to \tau_2$ then v is an abstraction.
3. if $\tau = [\tau]$ then v is an array of values.
4. if $\tau = \left(\vec{\tau}\right)$ then v is a tuple of values.

Lemma 3.2 (Weakening): If $\Gamma \vdash e : \tau$ and $x \notin \text{dom}(\Gamma)$, then $\Gamma, x : \tau' \vdash e : \tau$

Lemma 3.3 (Preservation of types under substitution): If $\Gamma, x : \tau \vdash e : \tau'$ and $\Gamma \vdash s : \tau$, then $\Gamma \vdash e\{x \mapsto s\} : \tau'$

3.2.2 Proving Soundness

We have split the proof of Theorem 3.1 into two parts; one for preservation and one for progress. As the whole proof is quite long we will in this section only showcase some of the cases for the proof. The full proof of preservation and progress can be found in Appendix B.4 and Appendix B.5, respectively.

Proof of Preservation

We will prove preservation by induction on the derivation of $\Gamma \vdash e : \tau$ using case analysis on the last rule in the derivation.

(**BT-If**): Then $e = \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$ and $e_1 : \textbf{Int}$ and $e_2 : \tau$ and $e_3 : \tau$. By assuming $e \to e'$ exists, we then derive the following applicable reduction rules:

(**B-Ift**): then $e' = e_2$ and from the typing of e_2 we get $\Gamma \vdash e' : \tau$.

(**B-Iff**): then $e' = e_3$ and from the typing of e_3 we get $\Gamma \vdash e' : \tau$.

(**B-If**): then we get $e' = \texttt{if}\ e_1'\ \texttt{then}\ e_2\ \texttt{else}\ e_3$, where $e_1 \to e_1'$. Using the inductive hypothesis we get that $\Gamma \vdash e_1' : \textbf{Int}$, therefore using (BT-If) we get $\Gamma \vdash e' : \tau$.

(**BT-App**): Then $e = e_1 e_2$, and $\Gamma \vdash e_1 : \tau_1 \to \tau$ and $\Gamma \vdash e_2 : \tau_1$. By assuming $e \to e'$ exists, and using case analysis we derive the following applicable reduction rules:

...:

(**B-Abs**): Then $e_1 = \lambda(x : \tau_2).e_3$ and $e_2 = v$ and $e' = e_3\{x \mapsto v\}$. Because $\Gamma \vdash e_1 : \tau_1 \to \tau$ and $e_1 = \lambda(x : \tau_2).e_3$ by inspection of the type rules it must hold that $\tau_1 = \tau_2$ giving us $\Gamma \vdash \lambda(x : \tau_1).e_3 : \tau_1 \to \tau$. Then by inspection the derivation must end with (BT-Abs) giving us $\Gamma, (x : \tau_1)e_3 : \tau$. Then because $\Gamma \vdash e_2 : \tau_1$ and $e_2 = v$ it must hold that $\Gamma \vdash v : \tau_1$. Therefore, using the Lemma 3.3 we have that $\Gamma \vdash e_3\{x \mapsto v\} : \tau$.

...:

Proof of Progress

We will prove progress by induction on the typing derivation of $\vdash e : \tau$ (the empty environment).

(**BT-If**): We know from (BT-If) that e_1 has type **Int**. By the inductive hypothesis we know that either $e_1 \in \mathcal{V}$ or $\exists e_1'.e_1 \to e_1'$. This gives us two cases:

- $e_1 \notin \mathcal{V}$
- $e_1 \in \mathcal{V}$

$\boldsymbol{e_1 \notin \mathcal{V}}$: In the case that $e_1 \notin \mathcal{V}$ then by our inductive hypothesis $e_1 \to e_1'$. We can then apply (B-If). We can see that given the premise we can take the step $\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \to \texttt{if}\ e_1'\ \texttt{then}\ e_2\ \texttt{else}\ e_3$ and therefore progress holds.

$\boldsymbol{e_1 \in \mathcal{V}}$: In the case that $e_1 \in \mathcal{V}$ then it must be number (that is the only value of type **Int** by proof of Lemma 3.1). In that case one of the two following rules applies: (B-Ift) or (B-Iff).

First case we apply (B-Ift) where given the premise e can take the step $\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \to e_2$ and therefore progress holds.

Second case we apply (B-Iff) where given the premise e can take the step $\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \to e_3$ and therefore progress holds.

(**BT-App**): We know by (BT-App) that e_1 has type $\tau_1 \to \tau_2$. By the induction hypothesis we know that $e_1 \in \mathcal{V}$ or $\exists e_1'.e_1 \to e_1'$ and $e_2 \in \mathcal{V}$ or $\exists e_2'.e_2 \to e_2'$. That gives us four cases:

- $e_1, e_2 \notin \mathcal{V}$
- $e_1, e_2 \in \mathcal{V}$
- $e_1 \in \mathcal{V}$ and $e_2 \notin \mathcal{V}$
- $e_2 \in \mathcal{V}$ and $e_1 \notin \mathcal{V}$.

$\boldsymbol{e_1, e_2 \notin \mathcal{V}}$: In the case that e_1 and e_2 are not values then by our inductive hypothesis $e_1 \to e_1'$ and $e_2 \to e_2'$. If e_1 takes a step then (B-App1) applies. From (B-App1) we have that given the premise we can take the step $e_1e_2 \to e_1'e_2$ and therefore progress holds.

if e_2 takes a step then (B-App2) applies. From (B-App2) we have that given the premise we can take the step $e_1e_2 \to e_1e_2'$ and therefore progress holds.

$\boldsymbol{e_1 \notin \mathcal{V}}$: In the case that e_1 is not a value then by our inductive hypothesis it can take a step. Then the rule (B-App1) applies. From (B-App1) we have that given the premise we can take the step $e_1e_2 \to e_1'e_2$ and therefore progress holds.

$\boldsymbol{e_2 \notin \mathcal{V}}$: In the case that e_2 is not a value then by our inductive hypothesis it can take a step. Then the rule (B-App2) applies. From (B-App2) we have that given the premise we can take the step $e_1 e_2 \to e_1 e_2'$ and therefore progress holds.

$\boldsymbol{e_1, e_2 \in \mathcal{V}}$: In the last case both e_1 and e_2 are values. As e_1 must be an arrow type (by our type rule) - then abstraction applies (as that is the only value that has type $\tau_1 \to \tau_2$ from proof of Lemma 3.1) and we can use (B-Abs) to take a step.

3.3 Typed Eπ

We will in this section extend Eπ with a type system (we call it TEπ). In creating a type system for TEπ, different approaches can be taken as many different variations of type systems for the π-calculus exists, including simple types, linear types, and session types [26]. We have chosen to start with designing a simple type system which can act as a base for future extensions.

3.3.1 Syntax and Semantics of TEπ

The formation rules of TEπ are a bit different from Eπ. First we have removed the labels `all`, `len` and `tup`. As these labels are just names to help with readability in the translation and do not have different semantics from $x \cdot y$ we have removed them. In addition we will restrict composite names to be only on variables and numbers. By doing this we can then change the syntax for composite names from $c \cdot l$ to $c \cdot T$ and use the type system to restrict composite names to only x and n in the type rules. The other difference is the type annotation on the name in restriction and the receive action.

$$
\begin{aligned}
P, Q, R, ... ::= {} & \mathbf{0} \\
& P \mid Q \\
& !P \\
& A.P \\
& \bullet P \\
& (\nu a : t).P \\
& [T_1 \bowtie T_2]P, Q
\end{aligned}
\qquad\qquad
\begin{aligned}
A ::= {} & \overline{c}\langle \vec{T} \rangle \\
& c\left(\vec{x} : \vec{t}\right) \\
& \overline{c}{:}\langle \vec{T} \rangle
\end{aligned}
$$

$$
\begin{aligned}
c ::= {} & a \\
& x \\
& c \cdot T
\end{aligned}
\qquad\qquad
\begin{aligned}
T ::= {} & n \\
& a \\
& x \\
& T \odot T
\end{aligned}
$$

Figure 3.3: Syntax for TEπ

The semantics of TEπ is the same as Eπ but with added type annotations as seen in Figure 3.3.

3.3.2 Type System

The type system for TEπ is inspired by the simple type system as introduced by Gay in [26] and elements of the $D\pi$ type-system from [27] to handle composite names. Like in BtF we have the primitive type **Int** and just like in the simple type system from [26], we have the channel type which indicate the type of object the channel carries.

Definition 3.4 (TEπ types):

$$t \coloneqq \mathbf{Int} \mid \mathrm{ch}\left(\vec{t}\right) \mid @\ell \mid \mathrm{pch}\left(\vec{t}\right)$$
$$\ell \coloneqq \{t_1, ..., t_n\}$$

To handle communication on composite names we introduce two types: $@\ell$ called a location type and $\mathrm{pch}\left(\vec{t}\right)$ called a pre-channel type. The location type is inspired by [28] as it contains a set of types that can be communicated on. However as we do not have sub-typing as in [28] we use composed names to communicate, where the type of the composed name should match with a type in the the set of types in the location type.

The type judgements for TEπ is of the following form: $\Delta, \Pi \vdash P$, where Δ, Π are type environments and P is a TEπ process. The Δ environment gives us the types of the free variables, names and natural numbers in P. The Π environment maps the location handle and the composed name or number to a pre-channel type ensuring that each array/tuple location handle has their own typing for the composed name. This then prevents out of bounds indexing as any pair (h, n) where n is greater than the size of the array should not map to a type in the environment Π.

Definition 3.5 (Type environment): A type environment Δ is partial function from free names and variables to types

$$\mathcal{N} \rightharpoonup \boldsymbol{\mathcal{T}}$$

Definition 3.6 (Pre-channel type environment): A composite name type environment Π is partial function from a pair of a name and a name/number to pre-channel types

$$(\mathcal{N}, \mathcal{N} \cup \mathbb{N}) \rightharpoonup \boldsymbol{\mathcal{T}}$$

To use these environments we have defined a simple notation for extending them with a name and its corresponding type (Definition 3.7). When extending with multiple names we may use $\Delta, \vec{u} : \vec{t}$ or $\Pi, \left(\vec{u}, \vec{n}\right) : \vec{t}$.

Definition 3.7 (Type environment extension): When extending a type environment with a name u or number n and a type t we write $\Delta, u : t$ or $\Pi, (u, n) : t$.

In addition we define the well-formed environment as follows.

Definition 3.8 (Well-formed Δ, Π): The type environments Δ, Π are well formed, if $\Delta, \Pi \vdash P$ and $\mathrm{fv}(P) \cup \mathrm{fn}(P) \subseteq \mathrm{dom}(\Delta) \cup \mathrm{dom}(\Pi)$.

We have split the type rules into two parts: type rules for processes and type rules for terms.

$$\text{(ET-Nil)}\ \frac{}{\Delta, \Pi \vdash \mathbf{0}} \qquad \text{(ET-Par)}\ \frac{\Delta, \Pi \vdash P \quad \Delta, \Pi \vdash Q}{\Delta, \Pi \vdash P \mid Q}$$

$$\text{(ET-Rep)}\ \frac{\Delta, \Pi \vdash P}{\Delta, \Pi \vdash !P} \qquad \text{(ET-Res)}\ \frac{\Delta, a : t, \Pi \vdash P}{\Delta, \Pi \vdash (\nu a : t).P}$$

$$\text{(ET-Send)}\ \frac{\Delta, \Pi \vdash c : \mathrm{ch}\left(\vec{t}\right) \quad \Delta, \Pi \vdash \vec{T} : t \quad \Delta, \Pi \vdash P}{\Delta, \Pi \vdash \overline{c}\langle \vec{T} \rangle.P} \qquad \text{(ET-Recv)}\ \frac{\Delta, \Pi \vdash c : \mathrm{ch}\left(\vec{t}\right) \quad \Delta, \vec{x} : t, \Pi \vdash P}{\Delta, \Pi \vdash c\left(\vec{x} : t\right).P}$$

$$\text{(ET-Broad)}\ \frac{\Delta, \Pi \vdash c : \mathrm{ch}\left(\vec{t}\right) \quad \Delta, \Pi \vdash \vec{T} : t \quad \Delta, \Pi \vdash P}{\Delta, \Pi \vdash \overline{c}{:}\langle \vec{T} \rangle.P} \qquad \text{(ET-Match)}\ \frac{\Delta, \Pi \vdash T_1 : t \quad \Delta, \Pi \vdash T_2 : t \quad \Delta, \Pi \vdash P \quad \Delta, \Pi \vdash Q}{\Delta, \Pi \vdash [T_1 \bowtie T_2]P, Q}$$

Figure 3.4: Type rules for TEπ processes

Type rule (ET-Par) is typed as usual by typing each process under the environment. (ET-Match) is typed similarly but we also require the two terms we match to be the same type. For restriction, (ET-Res), we require that the continuing process is well-typed with the restricted name added.

For (ET-Recv), we require the name we receive on to be a channel type, and that we can add the received names and their types to the environment and still be well-typed. There is no difference between the type rules of (ET-Send) and (ET-Broad). Like (ET-Recv), we require the name we are broadcasting/sending over, to be a channel type. The names we are sending/ broadcasting must also be the same type and the continuing process must also be well-typed.

$$\text{(ET-N)}\ \frac{}{\Delta, \Pi \vdash n : \mathbf{Int}} \qquad \text{(ET-U)}\ \frac{\Delta(u) = t}{\Delta, \Pi \vdash u : t}$$

$$\text{(ET-Bin)}\ \frac{\Delta, \Pi \vdash T_1 : \mathbf{Int} \quad \Delta, \Pi \vdash T_2 : \mathbf{Int}}{\Delta, \Pi \vdash T_1 \odot T_2 : \mathbf{Int}} \qquad \text{(ET-Compx)}\ \frac{\Delta \vdash u : @\ell \quad \Pi(u, x) = \text{pch}\left(\vec{t}\right) \quad \text{pch}\left(\vec{t}\right) \in \ell}{\Delta, \Pi \vdash u \cdot x : \text{ch}\left(\vec{t}\right)}$$

$$\text{(ET-Compn)}\ \frac{\Delta \vdash u : @\ell \quad \Pi(u, n) = \text{pch}\left(\vec{t}\right) \quad \text{pch}\left(\vec{t}\right) \in \ell}{\Delta, \Pi \vdash u \cdot n : \text{ch}\left(\vec{t}\right)}$$

Figure 3.5: Type rules for TEπ terms

We have one look-up rule (ET-U). Remember we use u when we do not differentiate between x and a. (ET-Bin) require the terms for the binary operation to have type **Int**. To handle composite names we have two type rules (ET-Compx) and (ET-Compn). First we require u to be a location type as we then can look in the Π-environment to see if the composition of the names has a pre-channel type. Then we ensure that the pre-channel type exists in the location type and therefore is valid name in the location.

3.4 Soundness of TEπ

Before giving the soundness theorem of TEπ we will introduce a *error* predicate. The *error* predicate is a process that should only be reached in case there is a type mismatch. In Figure 3.6, a subset of the reductions rules for the *error* predicate can be found. For all the rules see Appendix C.

$$\text{(ER-Broad)}\ \frac{\Delta(c) = \text{ch}\left(\vec{t}_1\right) \quad \Delta, \Pi \vdash \vec{T} : \vec{t}_2 \quad \vec{t}_1 \neq \vec{t}_2}{\Delta, \Pi \vdash \overline{c}{:}\langle\vec{T}\rangle.P \xrightarrow{:c} error} \qquad \text{(ER-Recv)}\ \frac{\Delta(c) = \text{ch}\left(\vec{t}_1\right) \quad \vec{t}_1 \neq \vec{t}_2}{\Delta, \Pi \vdash c\left(\vec{x} : \vec{t}_2\right).P \xrightarrow{\tau} error}$$

$$\text{(ER-Match)}\ \frac{\Delta, \Pi \vdash T_1 : t_1 \quad \Delta, \Pi \vdash T_2 : t_2 \quad t_1 \neq t_2}{\Delta, \Pi \vdash [T_1 \bowtie T_2]P, Q \xrightarrow{\tau} error}$$

Figure 3.6: Subset of the *error* predicate rules

(ER-Broad) can go to *error* in the case that the channel type and the type of the terms being sent does not match. (ER-Recv) is similar as we can go to *error* if the type of variables received do not match with type the channel type carries. (ER-Match) can go to *error* if the type of the terms being matched is not the same.

3.4.1 Soundness Theorem

With the *error* predicate introduced we can now give the soundness theorem for TEπ.

Theorem 3.2 (Soundness of TEπ): Let P be a TEπ process then
1. (Subject reduction) if $\Delta, \Pi \vdash P$ and $P \rightarrow P'$ then $\Delta, \Pi \vdash P'$
2. (Type safety) if $\Delta, \Pi \vdash P$ then $P \nrightarrow error$

For Theorem 3.2 we have two parts that needs to be proven. Subject reduction states that if a process P is well-typed in an environment and can reduce to P' then P' is also well-typed. Type safety states that if a process P is well-typed it can never reduce to *error*.

3.4.2 Proof of lemmas

Before we start the proof of Theorem 3.2 we need to define some lemmas which will help in the proof.

Lemma 3.4 (Weakening of terms): If $\Delta, \Pi \vdash T : t$ then $\Delta, u : t_u, \Pi \vdash T : t$ given any type t_u and a u such that $u \notin \text{dom}(\Delta)$.

Lemma 3.5 (Weakening of processes): If $\Delta, \Pi \vdash P$ then $\Delta, u : t_u, \Pi \vdash P$ given any type t_u and a u such that $u \notin \text{dom}(\Delta)$.

Lemma 3.6 (Strengthening of terms): If $\Delta, u : t_u, \Pi \vdash T : t$ then $\Delta, \Pi \vdash T : t$ given any type t_u and a u such that $u \notin \text{dom}(\Delta)$.

Lemma 3.7 (Strengthening of processes): If $\Delta, u : t_u, \Pi \vdash P$ then $\Delta, \Pi \vdash P$ given any type t_u and a u such that $u \notin \text{dom}(\Delta)$.

Lemma 3.4 to Lemma 3.7 states that we can add and remove names from the type environment where some name u is not used in some process P. This is used to prove Lemma 3.8 as in structural congruence we have the ability to extend the scope, add and remove restrictions. The full proof of all the lemmas can be found in Appendix D.1 to Appendix D.4.

Lemma 3.8 (Type preservation under structural congruence): Let P and Q be TE π processes.
1. If $\Delta, \Pi \vdash P$ and $P \equiv Q$ then $\Delta, \Pi \vdash Q$
2. If $\Delta, \Pi \vdash Q$ and $Q \equiv P$ then $\Delta, \Pi \vdash P$

Lemma 3.8 states that types are preserved under structural congruence. This lemma is necessary for proving subject reduction in Theorem 3.2. The full proof can be found in Appendix D.5.

Lemma 3.9 (Type preservation under substitution of processes): Let $\Delta, \Pi \vdash P$ be a well-typed TEπ process and $\Delta, \Pi \vdash T_u : t_u$, $\Delta, \Pi \vdash T'_u : t_u$, then $\Delta, \Pi \vdash P\left\{{}^{T'_u}/_{T_u}\right\}$

Lemma 3.10 (Type preservation under substitution of terms): If $\Delta, \Pi \vdash T : t$, $\Delta, \Pi \vdash T_u : t_u$ and $\Delta, \Pi \vdash T'_u : t_u$ such that $\Delta, \Pi \vdash T\left\{{}^{T_{u'}}/_{T_u}\right\} : t$

The last two lemmas, Lemma 3.9 and Lemma 3.10, states that types are preserved under substitution.

3.4.3 Proving Soundness

Just as the proof for Theorem 3.1, we have split the proof into two parts; one for subject reduction and one for type safety. As the whole proof is quite long we will in this section only showcase some of the cases for the proof. The full proof of subject reduction and type safety can be found in Appendix D.8 and Appendix D.9, respectively.

Proof of Subject Reduction

We will prove subject reduction by induction in the rule for concluding $P \to P'$.

(**E-Com**): By our assumption we know that $\Delta, \Pi \vdash \overline{c}\langle\vec{T}\rangle.P$ and $\Delta, \Pi \vdash c\left(\vec{x} : \vec{t}\right).Q$ by the application of (ET-Par). By (E-Com) we have $\overline{c}\langle\vec{T}\rangle.P | c\left(\vec{x}\right).Q \xrightarrow{\tau} P \mid Q\left\{{}^{\vec{T}}/_{\vec{x}}\right\}$ and must show that $\Delta, \Pi \vdash P \mid Q\left\{{}^{\vec{T}}/_{\vec{x}}\right\}$.

Then by (ET-Send) we have that $c : \mathrm{ch}\left(\vec{t}\right)$ and $\vec{T} : \vec{t}$, and by (ET-Recv) we have that $c : \mathrm{ch}\left(\vec{t}\right)$ and $\Delta, \vec{x} : \vec{t}, \Pi \vdash Q$. By using Lemma 3.9 we have that $\Delta, \vec{x} : \vec{t}, \Pi \vdash Q\left\{{}^{\vec{T}}/_{\vec{x}}\right\}$, and by Lemma 3.7 we have $\Delta, \Pi \vdash Q\left\{{}^{\vec{T}}/_{\vec{x}}\right\}$. We can therefore conclude $\Delta, \Pi \vdash P \mid Q\left\{{}^{\vec{T}}/_{\vec{x}}\right\}$ by (ET-Par).

(**E-Then**): By our assumption we know $\Delta, \Pi \vdash [T_1 \bowtie T_2]P, Q$ by (ET-Match) and $[T_1 \bowtie T_2]P, Q \xrightarrow{\tau} P$ by (E-Then). We must then show $\Delta, \Pi \vdash P$. This follows immediately by (ET-Match) as $\Delta, \Pi \vdash [T_1 \bowtie T_2]P, Q$ is only correct if $\Delta, \Pi \vdash P$.

(**E-Else**): By our assumption we know $\Delta, \Pi \vdash [T_1 \bowtie T_2]P, Q$ by (ET-Match) and $[T_1 \bowtie T_2]P, Q \xrightarrow{\tau} P$ by (E-Else). We must then show $\Delta, \Pi \vdash Q$. This follows immediately by (ET-Match) as $\Delta, \Pi \vdash [T_1 \bowtie T_2]P, Q$ is only correct if $\Delta, \Pi \vdash Q$.

Proof of Type Safety

We prove type safety by induction in the type rules.

(**ET-Broad**): By our assumption we have that $\Delta, \Pi \vdash \overline{c}{:}\langle \vec{T} \rangle.P$ and must prove that $\overline{c}{:}\langle \vec{T} \rangle.P \overset{:c}{\nrightarrow} \mathit{error}$. By inspecting (ER-Broad) we can see that $\overline{c}{:}\langle \vec{T} \rangle.P \overset{:c}{\longrightarrow} \mathit{error}$ only if $c : \mathrm{ch}\left(\vec{t_1}\right)$, $\vec{T} : \vec{t_2}$ where $\vec{t_1} \neq \vec{t_2}$. This contradicts the type rule for broadcast, that states $c : \mathrm{ch}\left(\vec{t}\right)$ and $\vec{T} : \vec{t}$ meaning it must hold that $\vec{t_1} = \vec{t_2}$ for broadcast to be well-typed and therefore $P \nrightarrow \mathit{error}$.

(**ET-Recv**): By our assumption we have that $\Delta, \Pi \vdash c\left(\vec{x} : \vec{t}\right).P$ and must prove that $c\left(\vec{x} : \vec{t}\right).P \overset{\tau}{\nrightarrow} \mathit{error}$. By inspecting (ER-Send) we can see that $c\left(\vec{x} : \vec{t}\right).P \overset{\tau}{\longrightarrow} \mathit{error}$ if $c : \mathrm{ch}\left(\vec{t_1}\right)$ and $\vec{x} : \vec{t_2}$ where $\vec{t_1} \neq \vec{t_2}$. This contradicts (ET-Recv) as for send to be well-typed $c : \mathrm{ch}\left(\vec{t}\right)$ and $\vec{x} : \vec{t}$, and therefore it must be that $t_1 = t_2$. We can therefore conclude $P \nrightarrow \mathit{error}$.

(**ET-Match**): By our assumption we have that $\Delta, \Pi \vdash [T_1 \bowtie T_2]P, Q$ and must prove that $[T_1 \bowtie T_2]P, Q \overset{\tau}{\nrightarrow} \mathit{error}$. By inspecting (ER-Match) we can see that $[T_1 \bowtie T_2]P, Q \overset{\tau}{\longrightarrow} \mathit{error}$ if $T_1 : t_1$, $T_2 : t_2$ and $t_1 \neq t_2$. This contradicts (ET-Match) as $T_1 : t$ and $T_2 : t$, that being $t_1 = t_2$ for match to be well-typed and therefore $P \nrightarrow \mathit{error}$

4 Translation and Correctness

We will in this chapter introduce the translation of BtF to TEπ and show the typed versions of the examples given in Chapter 2.3.2. We then define the correctness of the translations and show a part of the proof.

4.1 Translation of BtF to TEπ

As the semantics of BtF and TEπ is not different from the semantics of ButF and Eπ, the translation of the expressions themselves has not changed much. For the cases where the translation has changed we will discuss the differences compared to the original introduced in Chapter 2.3.

4.1.1 Translation

Just as the introduction of the original translation from ButF to Eπ by Hüttel et. al, [2], we use certain characters to illustrate their value. For translating the types and the type environment we take inspiration from Sangiorgi and Walker and their translation of the typed λ-calculus [29].

When translating expressions we use the notation $[\![e]\!]^{\Gamma}_{o} = (\Delta, \Pi, P)$ to denote translating some well-typed expression $\Gamma \vdash e : \tau$ to $\Delta, \Pi \vdash P$. In the case where o is a fresh name in P we add the restriction on o such that $[\![e]\!]^{\Gamma}_{o} = (\Delta, \Pi, (\nu o : \mathrm{ch}([\![\tau]\!])).P)$. To simplify the definition of the translation when sub-expression are present we simply insert the P_n from the resulting tuple (Δ_n, Π_n, P_n) from the sub-expression and collect the environments $(\Delta_1 \cup ... \cup \Delta_n, \Pi_1 \cup ... \cup \Pi_n, P)$. We omit the tuple notation when no change is made to Δ and Π in the translation. When referencing Π_n in the translation, it corresponds to the Π from $[\![e_n]\!]^{\Gamma}_{o_n} = (\Delta, \Pi, P)$.

Translation of Basic Expressions

The translation of basic expression is similar to the one as seen in Chapter 2.3.

$$[\![x]\!]^{\Gamma}_{o} = \overline{o}\langle x \rangle \qquad \text{where } \Gamma \vdash x : \tau$$

$$[\![n]\!]^{\Gamma}_{o} = \overline{o}\langle n \rangle \qquad \text{where } \Gamma \vdash n : \mathbf{Int}$$

$$[\![\lambda(x : \tau_1).e_1]\!]^{\Gamma}_{o} = (\nu h : [\![\tau_1 \to \tau_2]\!]).(\overline{o}\langle h \rangle \mid !h(x : [\![\tau_1]\!], r : \mathrm{ch}([\![\tau_2]\!])).[\![e_1]\!]^{\Gamma}_{r}) \qquad \text{where } \Gamma \vdash \lambda(x : \tau_1).e : \tau_1 \to \tau_2,\ \Gamma \vdash e : \tau_2$$

$$[\![e_1 e_2]\!]^{\Gamma}_{o} = (\nu o_1 : \mathrm{ch}([\![\tau_1 \to \tau_2]\!])).(\nu o_2 : \mathrm{ch}([\![\tau_1]\!])).\big([\![e_1]\!]^{\Gamma}_{o_1} \mid [\![e_2]\!]^{\Gamma}_{o_2} \mid o_1(h : [\![\tau_1 \to \tau_2]\!]).o_2(v : [\![\tau_1]\!]). \bullet\, \overline{h}\langle v, o \rangle\big) \qquad \text{where } \Gamma \vdash e_1 : \tau_1 \to \tau_2,\ \Gamma \vdash e_2 : \tau_1$$

$$[\![e_1[e_2]]\!]^{\Gamma}_{o} = (\nu o_1 : \mathrm{ch}([\![[\tau]]\!])).(\nu o_2 : \mathrm{ch}([\![\mathbf{Int}]\!])).\big([\![e_1]\!]^{\Gamma}_{o_1} \mid [\![e_2]\!]^{\Gamma}_{o_2} \mid o_1(h : [\![[\tau]]\!]). o_2(i : [\![\mathbf{Int}]\!]).h \cdot i(v : [\![\tau]\!]). \bullet\, \overline{o}\langle v \rangle\big) \qquad \text{where } \Gamma \vdash e_1 : [\tau],\ \Gamma \vdash e_2 : \mathbf{Int}$$

$$
\begin{array}{lll}
[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]^\Gamma_o = & (\nu o_1 : \mathrm{ch}([\![\mathbf{Int}]\!])).\big([\![e_1]\!]^\Gamma_{o_1} \mid & \text{where} \\
& o_1(n : [\![\mathbf{Int}]\!]).\bullet [n \neq 0][\![e_2]\!]^\Gamma_o, [\![e_3]\!]^\Gamma_o\big) & \Gamma \vdash e_1 : \mathbf{Int} \\
& & \Gamma \vdash e_2 : \tau \\
& & \Gamma \vdash e_3 : \tau \\
[\![e_1 \odot e_2]\!]^\Gamma_o = & (\nu o_1 : \mathrm{ch}([\![\mathbf{Int}]\!])).(\nu o_2 : \mathrm{ch}([\![\mathbf{Int}]\!])). & \text{where} \\
& \big([\![e_1]\!]^\Gamma_{o_1} \mid [\![e_2]\!]^\Gamma_{o_2} \mid o_1(v_1 : [\![\mathbf{Int}]\!]). & \Gamma \vdash e_1 : \mathbf{Int} \\
& o_2(v_2 : [\![\mathbf{Int}]\!]).\bullet \overline{o}\langle v_1 \odot v_2\rangle\big) & \Gamma \vdash e_2 : \mathbf{Int}
\end{array}
$$

Figure 4.1: Translation of basic BtF expressions

Indexing is one the more interesting cases as that has been changed the most. First we have moved the important reduction to the output of the value as we have removed the match on i. In the original translation the match stops us from proceeding if the value is below 0 but not if above the number of elements in the array. As we will show in the proof of Theorem 4.2 removing the match does not change the correctness of the behaviour for the translation.

Translation of Arrays and Tuples

For the translation of arrays and tuples we need an updated version of the *Cell* process with its type annotation.

$$
\mathit{Cell}(h, i, v, t_v) = !h \cdot \texttt{all}\ (r : \mathrm{ch}(\mathbf{Int}, t_v)).\overline{r}\langle i, v\rangle \mid !\overline{h \cdot i}\langle v\rangle \tag{3}
$$

One of the changes in the *Cell* process is in the output on $\overline{h \cdot i}$. In [2], when output on $\overline{h \cdot i}$ both the value and index was sent. As only the value and not the index was needed to be sent we have removed it. When we construct the *Cell* process we require the type of the value located at the index.

$$
\begin{array}{lll}
[\![(e_1, ..., e_n)]\!]^\Gamma_o = & (\nu o_1 : \mathrm{ch}([\![\tau_1]\!])).....(\nu o_n : \mathrm{ch}([\![\tau_n]\!])). & \text{where} \\
& \big(\textstyle\prod_{i=1}^n [\![e_i]\!]^\Gamma_{o_i} \mid o_1(v_1 : [\![\tau_1]\!]). \ ... \ .o_n(v_n : [\![\tau_n]\!]). & \Gamma \vdash (e_1, ..., e_n) : (\tau_1, ..., \tau_n) \\
& (\nu h : [\![(\tau_1, ..., \tau_n)]\!]).\big(!\overline{h \cdot \texttt{tup}}\langle v_1, ..., v_n\rangle \mid \overline{o}\langle h\rangle\big)\big) & \forall i \in \{1, ..., n\}.\Gamma \vdash e_i : \tau_i \\
& & \Pi = (\Pi_1 \cup ... \cup \Pi_n) \\
& & , (h, \texttt{tup}) : \mathrm{pch}([\![\tau_1]\!], ..., [\![\tau_n]\!]) \\
[\![[e_1, ..., e_n]]\!]^\Gamma_o = & (\nu o_1 : \mathrm{ch}([\![\tau]\!])).....(\nu o_n : \mathrm{ch}([\![\tau]\!])).(\nu h : [\![[\tau]]\!]). & \text{where} \\
& \big(\textstyle\prod_{i=1}^n [\![e_i]\!]^\Gamma_{o_i} \mid o_1(v_1 : [\![\tau]\!]).....o_n(v_n : [\![\tau]\!]). & \Gamma \vdash [e_1, ..., e_n] : [\tau] \\
& \big(\textstyle\prod_{i=1}^n \mathit{Cell}(h, i-1, v_i, [\![\tau]\!]) \mid \overline{h \cdot \texttt{len}}\langle n\rangle \mid \overline{o}\langle h\rangle\big)\big) & \forall i \in \{1, ..., n\}.\Gamma \vdash e_i : \tau \\
& & \Pi = (\Pi_1 \cup ... \cup \Pi_n) \\
& & , (h, \texttt{len}) : \mathrm{pch}(\mathbf{Int}) \\
& & , (h, \texttt{all}) : \mathrm{pch}(\mathrm{ch}(\mathbf{Int}, \tau)) \\
& & , (h, 1) : \mathrm{pch}([\![\tau]\!]) \\
& & \qquad \vdots \\
& & , (h, n) : \mathrm{pch}([\![\tau]\!])
\end{array}
$$

Figure 4.2: Translation of BtF arrays and tuples

The biggest addition to translation of array and tuples is the construction of the Π to ensure the correct typing of composed names. Building Π for tuples are quite simple we only add one pair $(h, \texttt{tup})$ with a pre-channel type for it. Then for arrays we add a pair for $\texttt{all}, \texttt{len}$ and then a pair for each index in the array.

Translation of Types and Environments

$$\begin{aligned}
[\![\mathbf{Int}]\!] &= \mathbf{Int} \\
[\![[\tau]]\!] &= @\{\mathrm{pch}(\mathbf{Int}, [\![\tau]\!]), \mathrm{pch}([\![\tau]\!]), \mathrm{pch}(\mathbf{Int})\} \\
[\![(\tau_1, ..., \tau_n)]\!] &= @\{\mathrm{pch}([\![\tau_1]\!], ..., [\![\tau_n]\!])\} \\
[\![\tau_1 \to \tau_2]\!] &= \mathrm{ch}([\![\tau_1]\!], \mathrm{ch}([\![\tau_2]\!]))
\end{aligned}$$

Figure 4.3: Translation of basic BtF types

First major addition in the translation we have, is the addition of the translation of types. The translation of **Int** is straightforward as there is a one to one correspondence for this type. The translation of the array type is interesting as it depends on the translation of arrays. The reason we have a pre-channel type for arrays is that communication with arrays in TEπ is on composite names and therefore we must type it as such. We have three composite names to communicate with the array on: $h \cdot \texttt{all}$, $h \cdot i$ and $h \cdot \texttt{len}$. The type of $\texttt{all}, i$ and $\texttt{len}$ corresponds to what can be seen as $\vec{t}_1$ in (ET-Compx) or (ET-Compn). The translation of the tuple type is very similar as communication with tuples is on the $h \cdot \texttt{tup}$ channel. Lastly, the translation of the abstraction type is similar to the intuition of applying a function. The first type is the argument and the second is the return channel with the resulting type of the function application.

$$\begin{aligned}
[\![\emptyset]\!] &= \emptyset \\
[\![\Gamma, x : \tau]\!] &= [\![\Gamma]\!], x : [\![\tau]\!]
\end{aligned}$$

Figure 4.4: Translation of type environment

Translation of Array Operations

Similar to the *Cell* process we need to update the *Repeat* process with type annotations. In addition we need to send a value on d and for the translations using *Repeat* we need a restriction on d for it to be well-typed.

$$Repeat(s, r, d) = (\nu c : \mathbf{Int}).(!c(n : \mathbf{Int}).\big([n \geq 0](\overline{r}\langle n-1\rangle \mid \overline{c}\langle n-1\rangle), \overline{d}\langle 0\rangle\big) \mid \overline{c}\langle s\rangle) \qquad (4)$$

The translation of array operations are then as seen in Figure 4.5. The translation of `map` has been changed by removing the check to ensure f is a function as that is required for it to be well-typed.

$$\begin{aligned}
[\![\texttt{size}]\!]^{\Gamma}_{o} &= (\nu o_1 : \mathrm{ch}([\![[\tau]]\!])).\big([\![e_1]\!]^{\Gamma}_{o_1} \mid \\
&\quad o_1(h : [\![[\tau]]\!]).h \cdot \texttt{len}\ (n : \mathbf{Int}).\bullet\, \overline{o}\langle n\rangle\big)
\end{aligned}$$

where
$\Gamma \vdash \texttt{size}\ e_1 : \mathbf{Int}$
$\Gamma \vdash e_1 : [\tau]$

$$\begin{aligned}
[\![\texttt{iota}\ e_1]\!]^{\Gamma}_{o} &= (\nu o_1 : \mathrm{ch}([\![\mathbf{Int}]\!])).(\nu r : \mathrm{ch}(\mathbf{Int})).(\nu d : \mathrm{ch}(\mathbf{Int})). \\
&\quad (\nu h : @\{\mathrm{pch}(\mathbf{Int}, \mathbf{Int}), \mathrm{pch}(\mathbf{Int})\}). \\
&\quad \big([\![e_1]\!]^{\Gamma}_{o_1} \mid o_1(n : [\![\mathbf{Int}]\!]).\big(Repeat(n, r, d) \mid \\
&\quad !r(i : \mathbf{Int}).\ Cell(h, i, i, \mathbf{Int}) \mid \bullet\, d(_ : \mathbf{Int}). \\
&\quad \overline{o}\langle h\rangle \mid \overline{h \cdot \texttt{len}}\langle n\rangle\big)\big)
\end{aligned}$$

where
$\Gamma \vdash \texttt{iota}\ e_1 : [\mathbf{Int}]$
$\Gamma \vdash e_1 : \mathbf{Int}$

$$
\begin{aligned}
[\![\texttt{map}\ e_1]\!]_o^\Gamma \;=\; & (\nu o_1 : \mathrm{ch}(t_1)).(\nu h_1 : [\![[\tau_2]]\!]).\Big([\![e_1]\!]_{o_1} \mid \\
& o_1(h_2 : t_1).h_2 \cdot \texttt{tup}\ (f : [\![\tau_1 \to \tau_2]\!], h_3 : [\![[\tau_1]]\!]). \\
& h_3 \cdot \texttt{len}\ (n : \mathbf{Int}).(\nu h_4 : \mathrm{ch}(\mathbf{Int}, [\![\tau_1]\!])). \\
& \overline{h_3 \cdot \texttt{all}}{:}\langle h_4 \rangle.(\nu r_1 : \mathrm{ch}(\mathbf{Int})).(\nu d : \mathrm{ch}(\mathbf{Int})). \\
& \Big(\mathit{Repeat}(n, r_1, d) \mid\ !h_4(i : \mathbf{Int}, v_1 : [\![\tau_1]\!]). \\
& (\nu r_2 : \mathrm{ch}([\![\tau_2]\!])).\overline{f}\langle v_1, r_2 \rangle.r_2(v_2 : [\![\tau_2]\!]) \\
& .r_1(\_ : \mathbf{Int}).\ \mathit{Cell}(h_1, i, v_2, [\![\tau_2]\!]) \mid \\
& \bullet\, d(\_ : \mathbf{Int}).\overline{o}\langle h_1 \rangle \mid\ !\overline{h_1 \cdot \texttt{len}}\langle n \rangle \Big) \Big)
\end{aligned}
$$

where
$\Gamma \vdash e_1 : (\tau_1 \to \tau_2, [\tau_1])$
$t_1 = [\![(\tau_1 \to \tau_2, [\tau_1])]\!]$

Figure 4.5: Translation of BtF array operations

4.1.2 Examples of Translations

We will now show some examples of translations. We will take the same examples as shown in Chapter 2.3.2.

Example of Indexing

For the first example we have the indexing of an array like on page 11. The expression is as follows: $[2,3][1]$, with the translation of the expression being:

$$
\begin{aligned}
[\![[2,3][1]]\!]_o^\Gamma = {} & (\nu o_{\mathrm{arr}} : \mathrm{ch}([\![[\mathbf{Int}]]\!])).(\nu o_i : \mathrm{ch}(\mathbf{Int})). \\
& ((\nu o_2 : \mathrm{ch}(\mathbf{Int})).(\nu o_3 : \mathrm{ch}(\mathbf{Int})).(\nu h : [\![[\mathbf{Int}]]\!]). \\
& (\overbrace{\overline{o_2}\langle 2 \rangle \mid \overline{o_3}\langle 3 \rangle}^{[\![2]\!]^\Gamma_{o_2} \text{ and } [\![3]\!]^\Gamma_{o_3}} \mid \overbrace{o_2(v_2 : \mathbf{Int}).o_3(v_3 : \mathbf{Int}).}^{\text{receiver}} \\
& \underbrace{(\mathit{Cell}(h, 0, v_2, \mathbf{Int}) \mid \mathit{Cell}(h, 1, v_3, \mathbf{Int}) \mid \overline{h \cdot \texttt{len}}\langle 2 \rangle \mid \overline{o_{\mathrm{arr}}}\langle h \rangle))}_{\text{array: [2,3]}} \\
& \mid \overbrace{\overline{o_i}\langle 1 \rangle}^{\text{index}} \mid \underbrace{\overbrace{o_{\mathrm{arr}}(h : [\![[\mathbf{Int}]]\!]).o_i(i : \mathbf{Int}).}^{\text{receive array handle and index}} \overbrace{h \cdot i(v : \mathbf{Int})}^{\text{receive value at index}} .\bullet\, \overline{o}\langle v \rangle)}_{\text{access and output of value}}
\end{aligned}
$$

First we create restrictions on the different channels used for communication. These include channels for the array handle and the values in the array. Then we translate the respective values in the array and receive them on their output channel (marked as receiver). We then construct the array using the *Cell* process (marked as: array : $[2,3]$). The *Cell* process for the value at index 1 is a follow:

$$
\mathit{Cell}(h, 1, v_2, \mathbf{Int}) = h \cdot \texttt{all}\ (r : \mathrm{ch}(\mathbf{Int}, \mathbf{Int})).\overline{r}\langle 1, v_3 \rangle \mid\ !\overline{h \cdot 1}\langle v_3 \rangle;
$$

After the translation of the indexing number we receive the array handle and index. We can then communicate with the *Cell* process on $h \cdot 1$ to receive the value located at the index. Finally we can send the value on our output channel o.

Example of Abstraction

For the second example we have the simple abstraction shown on page 12. The expression is as follows $\lambda(x : \mathbf{Int}).x + 1$ and is translated as shown below.

$$[\![\lambda x : \mathbf{Int}.x + 1]\!]^\Gamma_o = (\nu h : [\![\mathbf{Int} \to \mathbf{Int}]\!]).(\overline{o}\langle h\rangle \mid \underbrace{!h(x : [\![\mathbf{Int}]\!], r : \mathtt{ch}\ [\![\mathbf{Int}]\!])}_{\text{argument and return channel}}.[\![x + 1]\!]^\Gamma_r$$

$$[\![x + 1]\!]^\Gamma_r = \underbrace{(\nu o_1 : \mathrm{ch}([\![\mathbf{Int}]\!])).(\nu o_2 : \mathrm{ch}([\![\mathbf{Int}]\!])).(\overbrace{\overline{o_1}\langle x\rangle \mid \overline{o_2}\langle 1\rangle}^{[\![x]\!]^\Gamma_{o_1} \text{ and } [\![1]\!]^\Gamma_{o_2}} \mid \overbrace{o_1(v_1 : \mathbf{Int}).o_2(v_2 : \mathbf{Int})}^{\text{receive values}}.\overbrace{\overline{r}\langle v_1 + v_2\rangle}^{\text{send result}})}_{[\![x+1]\!]^\Gamma_r}$$

First we create a restriction on h, the function handle with the type $[\![\mathbf{Int} \to \mathbf{Int}]\!]$, which is translated to $\mathrm{ch}(\mathbf{Int}, \mathrm{ch}(\mathbf{Int}))$. Then we send the handle on o in parallel with constructing the function. The construction of the body consists of receiving the argument and return channel x, r then translating the expressions x and 1, respectively, and receive them on the output channels o_1 and o_2. Finally we send the result of $v_1 + v_2$ over r.

Example of Map

In the last example we have the translation of an expression with `map` as seen on page 12. The expression is the following: `map` $(\lambda(x : [\mathbf{Int}])$. `size` $x, [[1, 2], [3, 4]])$. This translation is quite long and therefore we have split it into sub-translations.

First we have the translation of the two inner arrays.

$$\begin{aligned}[\![[1, 2]]\!]^\Gamma_{o_A} = {} & (\nu o_1 : \mathrm{ch}(\mathbf{Int})).(\nu o_2 : \mathrm{ch}(\mathbf{Int})).(\nu h_A : [\![[\mathbf{Int}]]\!]). \\ & (\overbrace{\overline{o_1}\langle 1\rangle \mid \overline{o_2}\langle 2\rangle}^{[\![1]\!]^\Gamma_{o_1} \text{ and } [\![2]\!]^\Gamma_{o_2}} \mid \overbrace{o_1(v_1 : \mathbf{Int}).o_2(v_2 : \mathbf{Int})}^{\text{receiver}}. \\ & (\underbrace{\mathit{Cell}(h_A, 0, v_1, \mathbf{Int}) \mid \mathit{Cell}(h_A, 1, v_2, \mathbf{Int}) \mid \overline{h_A \cdot \mathtt{len}}\langle 2\rangle \mid \overline{o_A}\langle h_A\rangle}_{\text{array: [1,2]}}))\end{aligned}$$

$$\begin{aligned}[\![[3, 4]]\!]^\Gamma_{o_B} = {} & (\nu o_3 : \mathrm{ch}(\mathbf{Int})).(\nu o_4 : \mathrm{ch}(\mathbf{Int})).(\nu h_B : [\![[\mathbf{Int}]]\!]). \\ & (\overbrace{\overline{o_3}\langle 3\rangle \mid \overline{o_4}\langle 4\rangle}^{[\![3]\!]^\Gamma_{o_3} \text{ and } [\![4]\!]^\Gamma_{o_4}} \mid \overbrace{o_3(v_3 : \mathbf{Int}).o_4(v_4 : \mathbf{Int})}^{\text{receiver}}. \\ & (\underbrace{\mathit{Cell}(h_B, 0, v_3, \mathbf{Int}) \mid \mathit{Cell}(h_B, 1, v_4, \mathbf{Int}) \mid \overline{h_B \cdot \mathtt{len}}\langle 2\rangle \mid \overline{o_B}\langle h_B\rangle}_{\text{array: [3,4]}}))\end{aligned}$$

This part of the translation is quite simple. First the values of the array are translated and then the array is created using the *Cell* process just like in the indexing example. Next we have the translation of the outer array. We create the array using the handles we received from the translation of the two inner arrays as the value on the index.

$$\begin{aligned}[\![[[1, 2], [3, 4]]]\!]^\Gamma_{o_{\text{arr}}} = {} & (\nu o_A : \mathrm{ch}([\![[\mathbf{Int}]]\!])).(\nu o_B : \mathrm{ch}([\![[\mathbf{Int}]]\!])).(\nu h : [\![[[\mathbf{Int}]]]\!]). \\ & ([\![[1, 2]]\!]^\Gamma_{o_A} \mid [\![[3, 4]]\!]^\Gamma_{o_B} \\ & \mid \overbrace{o_A(h_A : [\![[\mathbf{Int}]]\!]).o_B(h_B : [\![[\mathbf{Int}]]\!])}^{\text{receive handles to inner arrays}}. \\ & (\underbrace{\mathit{Cell}(h, 0, h_A, [\![[\mathbf{Int}]]\!]) \mid \mathit{Cell}(h, 1, h_B, [\![[\mathbf{Int}]]\!]) \mid \overline{h \cdot \mathtt{len}}\langle 2\rangle \mid \overline{o_{\text{arr}}}\langle h\rangle}_{\text{array: [[1,2],[3,4]]}}))\end{aligned}$$

Next we have the translation of the abstraction. We start by outputting the handle such that processes can communicate with the abstraction process. In parallel with this, we have a replicated input on the handle which waits for the argument and the return channel were we

output the result after applying the abstraction. This is followed by the translation of our sub-expression which is a direct translation of $\texttt{size}\ e_1$ with e_1 being x.

$$[\![\lambda(x : [\mathbf{Int}]).\ \texttt{size}\ x]\!]^{\Gamma}_{o_F} = (\nu h_F : [\![[\mathbf{Int}] \to \mathbf{Int}]\!]).(\overline{o_F}\langle h_F\rangle \mid \underbrace{!h_F(x : [\![[\mathbf{Int}]]\!], r : \mathrm{ch}(\mathbf{Int}))}_{\text{Argument \& return channel}}.$$

$$\underbrace{(\nu o_x : \mathrm{ch}([\![[\mathbf{Int}]]\!])).(\overbrace{\overline{o_x}\langle x\rangle}^{[\![x]\!]^{\Gamma}_{o_x}} \mid \overbrace{o_x(h_{\mathrm{arr}} : [\![[\mathbf{Int}]]\!]).h_{\mathrm{arr}} \cdot \texttt{len}\ (n : \mathbf{Int})}^{\text{Get size of array}}).\bullet\, \overline{r}\langle n\rangle}_{[\![\texttt{size}\ x]\!]^{\Gamma}_{r}}$$

In the translation of the tuple we first have the translation of the two sub-expressions which we have already shown. From their respective output channel we receive the function handle and array handle. We can then output the tuple handle such that other processes can communicate with it.

$$[\![(\lambda x.\texttt{size}\ x, [[1,2],[3,4]])]\!]^{\Gamma}_{o_{\texttt{tup}}} = (\nu o_F : \mathrm{ch}([\![[\mathbf{Int}] \to \mathbf{Int}]\!])).(\nu o_{\mathrm{arr}} : \mathrm{ch}([\![[[\mathbf{Int}]]]\!])).$$

$$\overbrace{[\![\lambda x.\ \texttt{size}\ x]\!]^{\Gamma}_{o_F} \mid [\![[[1,2],[3,4]]]\!]^{\Gamma}_{o_{\mathrm{arr}}}}^{\text{Tuple elements}}$$

$$\mid \underbrace{o_F(f).o_{\mathrm{arr}}(h_{\mathrm{in}})}_{\text{Function and array handle}}.(\nu h_{\texttt{tup}} : [\![([\mathbf{Int}] \to \mathbf{Int}, [[\mathbf{Int}]])]\!]).$$

$$\underbrace{(!\overline{h_{\texttt{tup}} \cdot \texttt{tup}}\langle f, h_{\mathrm{in}}\rangle \mid \overline{o_{\texttt{tup}}}\langle h_{\texttt{tup}}\rangle)}_{\text{Tuple communication}}$$

Lastly we have the translation of the mapping. First, from the translation of the tuple we get the handle which we can then use to unpack the tuple. From the input array in the tuple we get the length of the array which will be used later in the *Repeat* process (marked as Setup guard). Additionally we also get the elements from the array, that being the two sub-arrays. When we receive the array element we can then apply the function. With the value we receive after a function application we can create a cell using the *Cell* process for the resulting array. When we receive on d (marked guard) we know the *Repeat* process has finished and can then output the new array handle.

$$
\begin{aligned}
[\![\texttt{map}\ (\lambda x.\texttt{size}\ x, [[1,2],[3,4]])]\!]_o^\Gamma = {} & (\nu o_{\texttt{tup}} : \mathrm{ch}([\![([\mathbf{Int}] \to \mathbf{Int}, [[\mathbf{Int}]])]\!]).(\nu h_{\mathrm{out}} : [\![[\mathbf{Int}]]\!]). \\
& \underbrace{([\![(\lambda x.\texttt{size}\ x, [[1,2],[3,4]])]\!]^\Gamma_{o_{\texttt{tup}}} \mid o_{\texttt{tup}}(h_{\texttt{tup}} : [\![([\mathbf{Int}] \to \mathbf{Int}, [[\mathbf{Int}]])]\!])}_{\text{Tuple}}. \\
& \underbrace{h_{\texttt{tup}} \cdot \texttt{tup}\ (f : [\![[\mathbf{Int}] \to \mathbf{Int}]\!], h_{\mathrm{in}} : [\![[[\mathbf{Int}]]]\!])}_{\text{Unpack tuple}}. \\
& \underbrace{h_{\mathrm{in}} \cdot \texttt{len}\ (n : \mathbf{Int})}_{\text{Input array size}}.(\nu h_{\texttt{all}} : \mathrm{ch}(\mathbf{Int}, [\![[\mathbf{Int}]]\!])). \\
& \underbrace{\overline{h_{\mathrm{in}} \cdot \texttt{all}}{:}\langle h_{\texttt{all}} \rangle}_{\text{Get input array elements}}.\ (\nu r_{\mathrm{idx}} : \mathrm{ch}(\mathbf{Int})).(\nu d : \mathrm{ch}(\mathbf{Int})). \\
& (\underbrace{\mathit{Repeat}(n, r_{\mathrm{idx}}, d)}_{\text{Setup guard}} \mid \underbrace{!h_{\texttt{all}}(i : \mathbf{Int}, v_{\mathrm{sub}} : [\![[\mathbf{Int}]]\!])}_{\text{Receive input array element}}. \\
& (\nu r_{\mathrm{sub}} : \mathrm{ch}(\mathbf{Int})).\underbrace{\overline{f}\langle v_{\mathrm{sub}}, r_{\mathrm{sub}} \rangle.r_{\mathrm{sub}}(s_z : \mathbf{Int})}_{\text{apply function}}. \\
& \underbrace{r_{\mathrm{idx}}(\_ : \mathbf{Int}).\ \mathit{Cell}(h_{\mathrm{out}}, i, s_z, \mathbf{Int})}_{\text{Unguard new array element}} \mid \underbrace{\bullet\, d(\_ : \mathbf{Int})}_{\text{Guard}}. \\
& \overline{o}\langle h_{\mathrm{out}} \rangle \mid !\overline{h_{\mathrm{out}} \cdot \texttt{len}}\langle n \rangle))
\end{aligned}
$$

4.2 Correctness of the Translation

As we have now shown the translation of BtF to TEπ, the next important step is to show that the translation is correct. But before we get to the proof we will need to define what a correct translation is. Our approach to define and prove correctness is similar to [2], that being by an operational correspondence, with a few changes brought forth by the type systems.

4.2.1 Defining Bisimulation

We have some important definitions we need before we can define operational correspondence. One of the main parts of operational correspondence is the notion of bisimilarity - more specifically in our case, a barbed congruence. A barb, as introduced by Milner and Sangiorgi in [30], is a predicate $\downarrow_\alpha$ such that $P \xrightarrow{\alpha}$ that being P can perform some observable action α. In our case α is either an input (a), output ($\overline{a}$) or a broadcast ($\overline{a}$:).

For our definition of barbed bisimulation, we have the notion of multiple important and administrative reductions. The definition we give is the same as seen in [2].

Definition 4.1 (Multiple important and administrative transitions): The transition $\overset{s}{\Longrightarrow}$ is defined for labels $s \in \{\circ, \bullet\}$ as follows:

$$
\overset{s}{\Longrightarrow} = \begin{cases} s = \circ, \xrightarrow{\circ}{}^* \\ s = \bullet, \xrightarrow{\circ}{}^* \xrightarrow{\bullet} \end{cases}
$$

Another important definition we introduce is the complete context. This approach of constricting contexts is inspired by Sangiorgi, [31].

Definition 4.2 (Complete context): Let $\Delta, \Pi \vdash P$ and $C[\cdot]$ be a context with a hole with $\text{fn}(C[\cdot]) \cup \text{fv}(C[\cdot]) = \mathcal{M}$, then we say C is a complete context if $\exists \Delta', \Pi'$ s.t $\Delta', \Pi' \vdash C[P]$ where $\text{dom}(\Delta', \Pi') = \mathcal{M}$.

Now we can finally give the definition of Weak Administrative Barbed Bisimulation (WABB). The definition of WABB is similar to the one seen in [2].

Definition 4.3 (Weak Administrative Barbed Bisimulation): Suppose $\Delta, \Pi \vdash P, Q$ and let $\mathcal{R}$ be symmetric relation over processes. Then $\mathcal{R}$ is called a Weak Administrative Barbed Bisimulation if for whenever $P \,\mathcal{R}\, Q$ the following holds:

1. If $P \xrightarrow{\bullet} P'$ then $Q \overset{\bullet}{\Longrightarrow} Q'$ and $P' \,\mathcal{R}\, Q'$
2. If $P \xrightarrow{\circ} P'$ then $Q \overset{\circ}{\Longrightarrow} Q'$ and $P' \,\mathcal{R}\, Q'$
3. For each prefix α, $P \downarrow_\alpha$ implies $Q \overset{\circ}{\Longrightarrow} \downarrow_\alpha$
4. For all complete contexts C, $C(P) \,\mathcal{R}\, C(Q)$

We write $P \dot{\approx}_\alpha Q$ if there exist a weak administrative barbed bisimulation $\mathcal{R}$ such that $P \,\mathcal{R}\, Q$.

The definition of a complete context (Definition 4.2) is necessary in the definition of WABB. Let us imagine we defined WABB without a complete context and just allowed a relation on every context filled with well-typed processes. We would then be able to relate two process that by themselves are well-typed, but put in a context are ill-typed. The complete context restricts the contexts such that we only look at contexts that is also well-typed when the hole is filled with a well-typed process.

4.2.2 Defining Correctness

With the definition of WABB we can start defining correctness of the translation. We have two requirements for the translation. The first requirement is the correctness of types is preserved after translation. This gives us the following theorem.

Theorem 4.1 (Typed correctness): Let e be a BtF expression.

1. (Soundness) If $\Gamma \vdash e : \tau$ then $[\![e]\!]_o^\Gamma = (\Delta, \Pi, P)$ where $\Delta, \Pi \vdash P$ and $\Delta(o) = \text{ch}([\![\tau]\!])$.
2. (Completeness) If $[\![e]\!]_o^\Gamma = (\Delta, \Pi, P)$, $\Delta, \Pi \vdash P$ and there exists $\Delta(o) : \text{ch}(t)$ with $t = [\![\tau]\!]$ then $\Gamma \vdash e : \tau$.

We then say that $e \precsim [\![e]\!]_o^\Gamma$ if the typed translation is sound and complete.

The second requirements is the behaviour is preserved after the translation. The second part is achieved by the notion of operational correspondence which is inspired by Amadio et. al, [17]. Our definition of operational correspondence is similar to the one as seen in [2] with the addition of requiring that expressions and processes must be well-typed.

Definition 4.4 (Operational Correspondence): Let $\Gamma \vdash e : \tau$ be a well-typed BtF expression, $\Delta, \Pi \vdash P$ a well-typed TEπ process and R be non-symmetric binary relation between an expression and a process. Then R is an administrative operational correspondence if $\forall (e, P) \in R$ the following holds:

1. If $e \to e'$ then $\exists P'$ such that $P \overset{\bullet}{\Longrightarrow} Q$, $Q \dot{\approx}_{\alpha} P'$ and $e' \, R \, P'$
2. If $P \overset{\bullet}{\Longrightarrow} P'$ then $\exists e', Q$ such that $e \to e'$, $Q \dot{\approx}_{\alpha} P'$ and $e' \, R \, Q$

We say that $e \gtrsim_{\text{ok}} P$ if there exists an operational correspondence relation R such that $e \, R \, P$.

From the definition of operational correspondence we have two conditions. Condition 1 achieves soundness by guaranteeing that reductions of BtF expression e can be matched by a sequence of one or more reductions in the corresponding TEπ process P. Condition 2 achieves completeness by ensuring that e can always evolve to some e' given that P has any important reduction and that the evolved expression e' is in an operational correspondence with some Q, and that P' and Q are WABB. The later part is important for ensuring the operational correspondence after a reduction. For the behavioural correctness we get the following theorem.

Theorem 4.2 (Behavioural correctness): Let e be a well-typed BtF expression and o be a fresh name then $e \gtrsim_{\text{ok}} [\![e]\!]_o^{\Gamma}$.

Theorem 4.2 states that a BtF expression and the translation of the expression is in an operational correspondence and in the proof we will show this. In the proof we will denote condition 1 as soundness and condition 2 as completeness. To help prove Theorem 4.2 we need some lemmas as seen in [2].

Lemma 4.1 (Program behaviour): For any $\Delta, \Pi \vdash P$, complete context C and $s \in \{\circ, \bullet\}$, then if there exists a $\Delta, \Pi \vdash Q$ s.t $C[P] \xrightarrow{s} Q$, then one of the following holds:

1. Only C reduces, therefore $Q = C'[P]$ s.t $C[\mathbf{0}] \xrightarrow{s} C'[\mathbf{0}]$.
2. Only P reduces, therefore $Q = C[P']$ s.t $P \xrightarrow{s} P'$.
3. C and P interact, therefore $Q = C'[P']$ and there exists an O s.t $C[P] \equiv \left(\nu \vec{a} : \vec{t}_a\right).(O \mid P)$, $O \mid P \xrightarrow{s} O' \mid P'$ and $C[P] \xrightarrow{s} C'[P']$.

Lemma 4.1 states three different cases in which processes can reduce when within a context. In the first case only the context reduces. In the second only the process within the context reduces. In the last case an interaction occurs and as such both the context and process reduces. This lemma is useful for simplifying the proofs for some of the later lemmas.

Lemma 4.2 (Preservation of substitution): let $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$ be well-typed BtF expression and $e_2 \in \mathcal{V}$ then

1. If e_2 is a number then $[\![e_1]\!]_o^\Gamma\{^n/_x\} \dot{\approx}_\alpha [\![e_1\{x \mapsto n\}]\!]_o^\Gamma$ for some o
2. If e_2 is an abstraction, tuple or array then $(\nu h : t).\big(Q \mid [\![e_1]\!]_o^\Gamma\{^h/_x\}\big) \dot{\approx}_\alpha [\![e_1\{x \mapsto e_2\}]\!]_o^\Gamma$ for some o, where $[\![e_2]\!]_o^\Gamma \mid o(x : t).P \overset{\bullet}{\Longrightarrow} (\nu h : t).\big(Q \mid P\{^h/_x\}\big)$ and $t = [\![\tau]\!]$.

Lemma 4.2 is necessary in proving Theorem 4.2, more specifically application. As in the application we have two expression $e_1 e_2$ with the first being an abstraction. We need to know if we can substitute in with the value from the second expression. This is necessary for us argue that the translation after reductions can be matched with application in BtF after the transition step.

The next two lemmas will be necessary to show that processes that are finished in the translation can be removed without any other processes being affected. First, Lemma 4.3 states that processes bisimilar to the **0** processes will always be bisimilar even after possible reductions. Second, Lemma 4.4 states that we can remove theses processes without affecting other processes.

Lemma 4.3 (Garbage processes): For any $\Delta, \Pi \vdash P$ then if $P \dot{\approx}_\alpha \mathbf{0}$ and there exists P' such that $P \overset{s}{\Longrightarrow}{}^* P'$ then $\forall \alpha. P' \not\downarrow_\alpha$ and $P' \dot{\approx}_\alpha \mathbf{0}$.

Lemma 4.4 (Garbage collection): For any complete context C and any $\Delta, \Pi \vdash P$ where $P \dot{\approx}_\alpha \mathbf{0}$, then for all $\Delta, \Pi \vdash Q$ it holds that $P \mid Q \dot{\approx}_\alpha Q$.

Lemma 4.5 is necessary if we want to know if there is an observable action after some administrative reductions. This will useful in the proof for behavioral correctness to argue for the bisimilarity of two processes. Lemma 4.6 states that if a translated expression after some reductions eventually outputs on o then e is a value.

Lemma 4.5 (Translated value has observable output): If $\Gamma \vdash e : \tau$ and $e \in \mathcal{V}$ then there exists an $\Delta, \Pi \vdash P$, s.t $[\![e]\!]_o^\Gamma \overset{\circ}{\Longrightarrow} P$ and $P \downarrow_{\overline{o}}$.

Lemma 4.6 (Translated expression has observable ouput): There exists $\Gamma \vdash e : \tau$ and $\Delta, \Pi \vdash P$ s.t $[\![e]\!]_o^\Gamma \overset{\circ}{\Longrightarrow} P$ and $P \downarrow_{\overline{o}}$ then it holds that $e \in \mathcal{V}$.

From the two theorems about correctness of the translation we get following corollary which states that any given well-typed BtF expression can be translated and stay well-typed in TEπ while preserving the behaviour from BtF.

Corollary 4.1 (Correctness of the typed translation): Given a well-typed BtF expression $\Gamma \vdash e : \tau$ and a fresh o then $e \gtrsim_{\text{ok}} [\![e]\!]_o^\Gamma$ and $e \precsim [\![e]\!]_o^\Gamma$

4.2.3 Proof of Correctness

For the corollary to hold we must prove the two theorems: Theorem 4.1 and Theorem 4.2.

We start with a subset of the proof of Theorem 4.1. The full proof can be found in Appendix E.7.

Proof of Theorem 4.1.

We prove soundness by induction on the rules used for concluding e is well-typed, and for completeness we prove it by induction on the structure of e.

Application: For application we have $e = e_1 e_2$ and must prove the soundness and completeness of the translation.

Soundness: From the (BT-App) rule we know that $e : \tau_2$. From inspection of the translation we have an $h : [\![\tau_1 \to \tau_2]\!] = \text{ch}([\![\tau_1]\!], \text{ch}([\![\tau_2]\!]))$. From the translation we see that we send v, o on h and from that we then have $v : [\![\tau_1]\!]$ and $o : \text{ch}([\![\tau_2]\!])$. Therefore soundness hold.

Completeness: We have that $e = \lambda(x : \tau_1).e_1$ and $[\![e]\!]_o^\Gamma = (\Delta, \Pi, P)$ where

$$P = (\nu h : [\![\tau_1 \to \tau_2]\!]).(\overline{o}\langle h\rangle \mid !h(x : [\![\tau_1]\!], r : \text{ch}([\![\tau_2]\!])).[\![e_1]\!]_r^\Gamma)$$

We assume that $\Delta(o) = \text{ch}(t)$ where $t = [\![\tau]\!]$. By inspection of the translation we have that $\overline{o}\langle h\rangle$ where $h : [\![\tau_1 \to \tau_2]\!]$ which implies $\tau = \tau_1 \to \tau_2$. Then by the induction hypothesis we get that $[\![e_1]\!]_o^\Gamma = (\Delta', \Pi, P')$ where $\Delta' = \Delta, h : [\![\tau_1 \to \tau_2]\!], x : [\![\tau_1]\!], r : \text{ch}([\![\tau_2]\!])$, s.t $\Gamma, x : \tau_1 \vdash e_1 : \tau_2$. Therefore by (BT-Abs) we get that $\Gamma \vdash \lambda(x : \tau_1).e_1 : \tau_1 \to \tau_2$.

□

For Theorem 4.2 we will show the proof of behavioral correctness for application. The full proof of Theorem 4.2 can be found in Appendix E.8.

Proof of Theorem 4.2.

Let $\mathcal{B}$ be the set of all BtF programs and let R be the the following relation $R = \{(e, [\![e]\!]_o^\Gamma) \mid e \in \mathcal{B}, o \text{ fresh}\}$. We show that R is an administrative operational correspondences. As per Definition 4.4 we only consider pairs where $e \to e'$ and where $[\![e]\!]_o^\Gamma$ contains $\bullet$.

Application: For application we have $e = e_1 e_2$ and must prove that the two parts of operational correspondence hold.

Soundness: For the first part we have that if $e \to e'$ then there is a P' such that $[\![e]\!]_o^\Gamma \overset{\bullet}{\Longrightarrow} P'$ and $P' \dot{\approx}_\alpha [\![e']\!]_o^\Gamma$. There are two cases for when $e \to e'$: One when the sub-expressions can take a step and one when they cannot.

For the first case we have that there are two application rules for the sub-expressions e_1 and e_2 to take a step: (B-App1) and (B-App2). By our assumption that $(e, [\![e]\!]_o^\Gamma) \in R$ we have that $\left(e_1, [\![e_1]\!]_{o_1}^\Gamma\right) \in R$. The same holds for e_2. When $[\![e_1]\!]_{o_1}^\Gamma$ and $[\![e_2]\!]_{o_2}^\Gamma$ is unguarded we know that $[\![e]\!]_o^\Gamma \overset{\bullet}{\Longrightarrow} P'$ and $P' \dot{\approx}_\alpha [\![e']\!]_o^\Gamma$.

For the second case we have that neither e_1 and e_2 cannot take a step and by Lemma 4.6 we know that e_1 and e_2 must be values and therefore must be on the form $e_1 = \lambda(x : \tau_1).e_\lambda$ and $e_2 = v$. In this case we can take a step with (B-Abs) and this can be matched in the translation of application. As this case is more complicated we will show how the translation will match this. First the two sub-expression is evaluated and this will give us the process on the following form:

$$(\nu o_1 : \mathrm{ch}([\![\tau_1 \to \tau_2]\!])).(\nu o_2 : \mathrm{ch}([\![\tau_1]\!])).$$
$$\Bigg(\underbrace{(\nu h : [\![\tau_1 \to \tau_2]\!]).(\overline{o}\langle h\rangle \mid\ !h(x : [\![\tau_1]\!], r : \mathrm{ch}([\![\tau_2]\!])).[\![e_\lambda]\!]_r^\Gamma)}_{[\![e_1]\!]_{o_1}^\Gamma}$$
$$\mid \underbrace{(\nu v : [\![\tau_1]\!]).\overline{o_2}\langle v\rangle \mid S}_{[\![e_2]\!]_{o_2}^\Gamma} \mid o_1(h : [\![\tau_1 \to \tau_2]\!]).o_2(v : [\![\tau_1]\!]).\bullet\, \overline{h}\langle v, o\rangle \Bigg)$$

As the translation of e_1 is an abstraction it is substituted with the translation of abstraction. The translation of e_2 is substituted with a value ready to be sent on o_2 in parallel with a processes S that maintains the value. To not confuse the reader, the expression in the translation of abstraction has been renamed to e_λ. After communication on o_1 and o_2 happens the application will be on the following form:

$$(\nu h : [\![\tau_1 \to \tau_2]\!]).(\nu v : [\![\tau_1]\!]).\underbrace{!h(x : [\![\tau_1]\!], r : \mathrm{ch}([\![\tau_2]\!])).[\![e_\lambda]\!]_r^\Gamma}_{\text{Abstraction}} \mid S \mid \bullet\, \overline{h}\langle v, o\rangle$$

Now we can send on the function handle h and thus proceed to $[\![e_\lambda]\!]_r^\Gamma$ where r is the return channel substituted with the output channel o and value v. We denote this as the process H which then corresponds to $H = [\![e_\lambda]\!]_o^\Gamma\{{}^v/_x\}$. By Lemma 4.2 we have that this corresponds to $e_\lambda\{v \mapsto x\}$ which is our e'. Thereby we have the $[\![e]\!]_o^\Gamma \overset{\bullet}{\Longrightarrow} P'$ and $P' \dot{\approx}_\alpha [\![e']\!]_o^\Gamma$.

Completeness: For the second part we have that if $[\![e]\!]_o^\Gamma \overset{\bullet}{\Longrightarrow} P'$ then there is an e' such that $e \to e'$ and $P' \dot{\approx}_\alpha [\![e']\!]_o^\Gamma$. We have two cases: first if the important transition happens in either $[\![e_1]\!]_{o_1}^\Gamma$ or $[\![e_2]\!]_{o_2}^\Gamma$, or second when sending on the handle h.

In the first case we can select e' to be either $e_1' e_2$ or $e_1 e_2'$ depending on where the important transition happens and then by one of the two application rules we have $e \to e'$.

In the second case both $[\![e_1]\!]_{o_1}^\Gamma$ and $[\![e_2]\!]_{o_2}^\Gamma$ can send on their respective o after some administrative reductions. By Lemma 4.6 we know that e_1 and e_2 are values and as

such no important transition exists in those. We know that $[\![e_1]\!]^{\Gamma}_{o_1}$ is an abstraction and therefore by (B-Abs) we have that $e \to e'$.

□

5 Conclusion

In this report we have taken a look at the small language ButF which encompasses some of the core aspects of Futhark and Eπ, an extension to the π-calculus extended with broadcasting capabilities and composite names, and the translation of ButF to $E\pi$. We then extended both with a simple type system (called BtF and TEπ, respectively) and introduced an updated translation.

5.1 Results

For the type system for BtF we took inspiration from the simply typed lambda calculus with some of the types based on a simplified Futharks type system, such as arrays and tuple types. As the goal was to attempt to create a type system similar to the simple type system of Futhark, we took inspiration from [20] for the type rules. Since a number of the type rules for Futhark is created with unique types, we were unable to capture those exactly with the simple types of BtF. As such, our type rules is mostly similar in those cases with others inspired by the simply typed λ-calculus. To handle typing of array operations we introduced an implicit type context where we could look up the type of `map`, `iota` and `size`. We showed that BtF is sound (Theorem 3.1) in regards to the semantics by showing that we can always take a reduction from a well-typed expression that is not a value and still be well-typed after the reduction.

In the type system for TEπ we introduced a simple type-system with location and pre-channel types to handle composite names. Furthermore we introduce a composite name environment that maps a tuple of names (x, y) to types. This ensures that we can reuse the name y for multiple handles x when translating from BtF to TEπ. This allows for a common interface for accessing arrays and tuples $h \cdot \texttt{all}$, $h \cdot \texttt{len}$, $h \cdot n$ and $h \cdot \texttt{tup}$ without typing collisions simplifying the translation. Furthermore this prevents out of bounds indexing as the pair $(h, n+1)$ should not map to a type in Π for any array with the handle h and the length n. As in BtF we prove that the type system for TEπ is sound (Theorem 3.2) by proving that we can take a reduction from a well-typed process and still be well-typed.

We then defined an updated translation of BtF to TEπ with some of the constraints of the original translation removed, as they became unnecessary through guarantees of the type-system. Furthermore we have reduced the amount of allowed programs, for example preventing using binary operation on two abstractions by ensuring that the values in binary operations have to be typed as **Int**. We then prove type correctness (Theorem 4.1) which ensures the translation of any well-typed BtF expression results in a well-typed TEπ expression, with the type of the final output being a channel type $\text{ch}(\llbracket\tau\rrbracket)$ where $\llbracket\tau\rrbracket$ is the translated type of the BtF expression. This is followed by a proof of the behavioural correctness of the translation (Theorem 4.2) which ensures that there is an operational correspondence between e and $\llbracket e\rrbracket_o^\Gamma$ such that $e \gtrsim_{\text{ok}} \llbracket e\rrbracket_o^\Gamma$. We then argue that (Theorem 4.1) and Theorem 4.2 prove Corollary 4.1 such that any given well-typed BtF expression can be translated and stay well-typed in TEπ while preserving the behaviour. This then gives us a data-parallel implementation of BtF in TEπ that ensures the behaviour and typing is correct. This also provides a starting point for extending the type system with sized and unique types from Futhark.

5.2 Future Work

There are several directions this work can be extended. Though BtF is a step closer to Futhark compared to ButF by including a type system, there are still many interesting aspects and designs in Futhark we have not been able to capture. One such could be their unique types or constructs not yet introduced to BtF. Below we will discuss some of the possible directions future work could take.

5.2.1 From Array Operations to Functions

Changing the semantics of BtF to allow `size`, `iota` and `map` as abstractions would simplify some programs as the example on page 12. Furthermore it would simplify Theorem 4.1 as there is a direct correspondence between the typing of array operation types in BtF and the TEπ process where now the type in TEπ is the resulting type of the arrow type in BtF. For example, the type for `size` would be $((\mathbf{Int}, [\tau] \to \mathbf{Int}))$ instead of $\mathbf{Int}$.

5.2.2 `with` Construct

Introducing the `with` construct from Futhark would be an interesting path to expand the translation to TEπ. As allowing in place updates without side effects, would require preventing the use of the input array and any aliases of it, in both BtF and TEπ. This problem could be potentially be solved using an environment that contains unique handles similar to the concept in [31] where Sangiorgi introduces an environment for names that must be used exactly once. This would require adding an extra condition to communication type rules that ensures the handle is not in the environment. Another interesting solution could be adding uniqueness types from Futhark [25]. Ensuring that either solution works is going to be interesting for the case of Lemma 4.2 where there are multiple handles in the TEπ translation to the same array where none should be accessible after using `with` in the body of the function.

5.2.3 Sized Types

Introducing sized types as in [32] would further align BtF with Futhark by ensuring bound checks at the type level. This however would require some degree of polymorphism on the size of arrays at the type level to allow for the implementation of function such as `size`, `iota` and `map`. This could be achieved by using universal quantification as in [32] and could look as follows $\Gamma \vdash \texttt{map} : \forall n.(\tau_1 \to \tau_2, [\tau_1](n)) \to [\tau_2](n)$. This would require introducing a specialization type rule as below where a specific size can be selected for a given universal quantifier though this would require defining substitution on types.

$$(\text{BT-spec})\ \frac{n \in \mathbb{N} \qquad \Gamma(f) : \forall x.\tau_1 \to \tau_2}{\Gamma(f) : \tau_1\{^x/_n\} \to \tau_2\{^x/_n\}}$$

6 Bibliography

[1] L. Jensen, C. O. Paulsen, and J. J. Teule, "Translating Concepts of the Futhark Programming Language into an Extended pi-Calculus," 2023, [Online]. Available: https://futhark-lang.org/student-projects/pi-msc-thesis.pdf

[2] H. Hüttel, L. Jensen, C. O. Paulsen, and J. Teule, "Functional Array Programming in an Extended Pi-Calculus," *Electronic Proceedings in Theoretical Computer Science*, vol. 412, pp. 2–18, Nov. 2024, doi: 10.4204/eptcs.412.2.

[3] Nvidia, "CUDA Toolkit." [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[4] K. Group, "OpenCL for Parallel Programming of Heterogeneous Systems." [Online]. Available: https://www.khronos.org/opencl/

[5] S. Cook, *CUDA Programming : a Developer's Guide to Parallel Computing with GPUs*. Amsterdam ; Boston: Morgan Kaufmann, 2013.

[6] "The Furthark Programming Language." [Online]. Available: https://futhark-lang.org/

[7] T. Henriksen, N. G. W. Serup, M. Elsman, F. Henglein, and C. E. Oancea, "Futhark: purely functional GPU-programming with nested parallelism and in-place array updates," *SIGPLAN Not.*, vol. 52, no. 6, pp. 556–571, Jun. 2017, doi: 10.1145/3140587.3062354.

[8] R. Bird, "Algebraic Identities for Program Calculation," *Comput. J.*, vol. 32, pp. 122–126, 1989, doi: 10.1093/comjnl/32.2.122.

[9] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, I," *Information and Computation*, vol. 100, no. 1, pp. 1–40, 1992, doi: https://doi.org/10.1016/0890-5401(92)90008-4.

[10] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, II," *Information and Computation*, vol. 100, no. 1, pp. 41–77, 1992, doi: https://doi.org/10.1016/0890-5401(92)90009-5.

[11] R. Milner, "The Polyadic π-Calculus: a Tutorial", in *Logic and Algebra of Specification*, F. L. Bauer, W. Brauer, and H. Schwichtenberg, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 203–246. doi: https://doi.org/10.1007/978-3-642-58041-3_6.

[12] D. Sangiorgi, "From π-calculus to higher-order π-calculus — and back", in *TAPSOFT'93: Theory and Practice of Software Development*, M. C. Gaudel and J. P. Jouannaud, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 151–166. doi: https://doi.org/10.1007/3-540-56610-4_62.

[13] C. Ene and T. Muntean, "Expressiveness of point-to-point versus broadcast communications," in *Fundamentals of Computation Theory*, G. Ciobanu and G. Păun, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 258–268. doi: https://doi.org/10.1007/3-540-48321-7_21.

[14] R. Milner, "Functions as processes," in *Automata, Languages and Programming*, M. S. Paterson, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1990, pp. 167–180. doi: https://doi.org/10.1007/BFb0032030.

[15] D. Sangiorgi, "From λ to π; or, Rediscovering continuations", *Mathematical Structures in Computer Science*, vol. 9, no. 4, pp. 367–401, 1999, doi: 10.1017/S0960129599002881.

[16] K. Honda, N. Yoshida, and M. Berger, "Control in the *pi*-Calculus", in *Proc.~Fourth ACM-SIGPLAN Continuation Workshop (CW'04)*, 2004.

[17] R. M. Amadio, L. Leth, and B. Thomsen, "From a Concurrent Lambda-Calculus to the Pi-Calculus," in *Proceedings of the 10th International Symposium on Fundamentals of Computation Theory*, in FCT '95. Berlin, Heidelberg: Springer-Verlag, 1995, pp. 106–115.

[18] D. Walker, "π-Calculus semantics of object-oriented programming languages", in *Theoretical Aspects of Computer Software*, T. Ito and A. R. Meyer, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 532–547. doi: https://doi.org/10.1007/3-540-54415-1_63.

[19] T. Noll and C. K. Roy, "Modeling Erlang in the pi-calculus," in *Proceedings of the 2005 ACM SIGPLAN Workshop on Erlang*, in ERLANG '05. Tallinn, Estonia: Association for Computing Machinery, 2005, pp. 72–77. doi: 10.1145/1088361.1088375.

[20] T. Henriksen, "Design and Implementation of the Futhark Programming Language," Universitetsparken 5, 2100 KÃ¸benhavn, 2017.

[21] M. Abadi, B. Blanchet, and C. Fournet, "The Applied Pi Calculus: Mobile Values, New Names, and Secure Communication," *J. ACM*, vol. 65, no. 1, Oct. 2017, doi: 10.1145/3127586.

[22] L. Jensen, C. O. Paulsen, and J. J. Teule, "Constructs of the Futhark Programming Language Described in a pi-Calculus," 2023, [Online]. Available: https://kbdk-aub.primo.exlibrisgroup.com/discovery/search?query=any,contains,3be42cfd-46d8-4308-81fd-4ff3863d97b3&tab=ProjekterSpecialer&search_scope=Projekter&vid=45KBDK_AUB:DDPB&lang=da&offset=0

[23] M. Carbone and S. Maffeis, "On the Expressive Power of Polyadic Synchronisation in π-calculus," *Electronic Notes in Theoretical Computer Science*, vol. 68, no. 2, pp. 15–32, 2002, doi: https://doi.org/10.1016/S1571-0661(05)80361-5.

[24] A. Church, "A formulation of the simple theory of types," *Journal of Symbolic Logic*, vol. 5, no. 2, pp. 56–68, 1940, doi: 10.2307/2266170.

[25] "The Futhark Language." [Online]. Available: https://futhark-book.readthedocs.io/en/latest/language.html#basic-language-features

[26] S. Gay, "Some Type Systems for the Pi Calculus," p. , 2000.

[27] M. Hennessy and J. Riely, "Resource Access Control in Systems of Mobile Agents," *Information and Computation*, vol. 173, no. 1, pp. 82–120, 2002, doi: https://doi.org/10.1006/inco.2001.3089.

[28] M. Hennessy, *A Distributed Pi-Calculus*. Cambridge University Press, 2007.

[29] D. Sangiorgi and D. Walker, *PI-Calculus: A Theory of Mobile Processes*. USA: Cambridge University Press, 2001.

[30] R. Milner and D. Sangiorgi, "Barbed bisimulation," in *Automata, Languages and Programming*, W. Kuich, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 685–695. doi: https://doi.org/10.1007/3-540-55719-9_114.

[31] D. Sangiorgi, "The name discipline of uniform receptiveness," *Theoretical Computer Science*, vol. 221, no. 1, pp. 457–493, 1999, doi: https://doi.org/10.1016/S0304-3975(99)00040-7.

[32] L. Bailly, T. Henriksen, and M. Elsman, "Shape-Constrained Array Programming with Size-Dependent Types," in *Proceedings of the 11th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing*, in FHPNC 2023. Seattle, WA, USA: Association for Computing Machinery, 2023, p. 29. doi: 10.1145/3609024.3609412.

A Appendix for Preliminaries

A.1 ButF Definitions

Definition 1.1 (Free variables of ButF):

$$
\begin{aligned}
\mathrm{FV}(x) &= \{x\} \\
\mathrm{FV}(\lambda x.e) &= \mathrm{FV}(e) \setminus \{x\} \\
\mathrm{FV}(e_1 e_2) &= \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\
\mathrm{FV}(e_1 \odot e_2) &= \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\
\mathrm{FV}([e_1, ..., e_n]) &= \mathrm{FV}(e_1) \cup ... \cup \mathrm{FV}(e_n) \\
\mathrm{FV}((e_1, ..., e_n)) &= \mathrm{FV}(e_1) \cup ... \cup \mathrm{FV}(e_n) \\
\mathrm{FV}(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) &= \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \cup \mathrm{FV}(e_3) \\
\mathrm{FV}(\texttt{map } e_1) &= \mathrm{FV}(e_1) \\
\mathrm{FV}(\texttt{iota } e_1) &= \mathrm{FV}(e_1) \\
\mathrm{FV}(\texttt{size } e_1) &= \mathrm{FV}(e_1)
\end{aligned}
$$

Definition 1.2 (Substitution of ButF):

$$
\begin{aligned}
x\{x \mapsto s\} &= s \\
y\{x \mapsto s\} &= y && \text{if } y \neq x \\
(\lambda y.e_1)\{x \mapsto s\} &= \lambda y.e_1\{x \mapsto s\} && \text{if } y \neq x \text{ and } y \notin \mathrm{FV}(s) \\
(e_1 e_2)\{x \mapsto s\} &= e_1\{x \mapsto s\}\ e_2\{x \mapsto s\} \\
(e_1 \odot e_2)\{x \mapsto s\} &= e_1\{x \mapsto s\} \odot e_2\{x \mapsto s\} \\
e_1[e_2]\{x \mapsto s\} &= e_1\{x \mapsto s\}[e_2\{x \mapsto s\}] \\
[e_1, ..., e_n]\{x \mapsto s\} &= [e_1\{x \mapsto s\}, ..., e_n\{x \mapsto s\}] \\
(e_1, ..., e_n)\{x \mapsto s\} &= (e_1\{x \mapsto s\}, ..., e_n\{x \mapsto s\}) \\
\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3\{x \mapsto s\} &= \texttt{if } e_1\{x \mapsto s\} \texttt{ then} \\
&\quad\ e_2\{x \mapsto s\} \texttt{ else } e_3\{x \mapsto s\} \\
(\texttt{map } e)\{x \mapsto s\} &= \texttt{map } (e\{x \mapsto s\}) \\
(\texttt{iota } e)\{x \mapsto s\} &= \texttt{iota } (e\{x \mapsto s\}) \\
(\texttt{size } e)\{x \mapsto s\} &= \texttt{size } (e\{x \mapsto s\})
\end{aligned}
$$

A.2 ButF Semantics

$$\text{(B-Array)}\ \frac{\exists i \in \{1,...,n\}.e_i \to e_i'}{[e_1,...,e_i,...,e_n] \to [e_1,...,e_i',...e_n]}$$

$$\text{(B-Tuple)}\ \frac{\exists i \in \{1,...,n\}.e_i \to e_i'}{(e_1,...,e_i,...,e_n) \to (e_1,...,e_i',...e_n)}$$

$$\text{(B-Index1)}\ \frac{e_1 \to e_1'}{e_1[e_2] \to e_1'[e_2]} \qquad \text{(B-Index2)}\ \frac{e_2 \to e_2'}{e_1[e_2] \to e_1[e_2']}$$

$$\text{(B-Index)}\ \frac{\forall i \in \{1,...,n\}}{[v_1,...,v_n][i] \to v_i} \qquad \text{(B-Abs)}\ \frac{}{(\lambda x.e)v \to e\{x \mapsto v\}}$$

$$\text{(B-App1)}\ \frac{e_1 \to e_1'}{e_1 e_2 \to e_1' e_2} \qquad \text{(B-App2)}\ \frac{e_2 \to e_2'}{e_1 e_2 \to e_1 e_2'}$$

$$\text{(B-If)}\ \frac{e_1 \to e_1'}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \to \texttt{if}\ e_1'\ \texttt{then}\ e_2\ \texttt{else}\ e_3}$$

$$\text{(B-Ift)}\ \frac{v \neq 0}{\texttt{if}\ v\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \to e_2} \qquad \text{(B-Iff)}\ \frac{v = 0}{\texttt{if}\ v\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \to e_3}$$

$$\text{(B-Bin1)}\ \frac{e_1 \to e_1'}{e_1 \odot e_2 \to e_1' \odot e_2} \qquad \text{(B-Bin2)}\ \frac{e_2 \to e_2'}{e_1 \odot e_2 \to e_1 \odot e_2'}$$

$$\text{(B-Bin)}\ \frac{v_3 = v_1 \odot v_2}{v_1 \odot v_2 \to v_3} \qquad \text{(B-Iota)}\ \frac{}{\texttt{iota}\ n \to [0,1,...,n-1]}$$

$$\text{(B-Size)}\ \frac{}{\texttt{size}\ [v_1,...,v_n] \to n}$$

$$\text{(B-Map)}\ \frac{}{\texttt{map}(\lambda x.e, [v_1,...,v_n]) \to [e\{x \mapsto v_1\},...,e\{x \mapsto v_n\}]}$$

Figure 1.1: ButF semantics

A.3 Eπ Definitions

Definition 1.3 (Free names of channels):

$$\begin{aligned}\mathrm{fn}(a) &= \{a\}\\ \mathrm{fn}(x) &= \emptyset\\ \mathrm{fn}(a \cdot l) &= \{a\}\\ \mathrm{fn}(x \cdot l) &= \emptyset\end{aligned}$$

Definition 1.4 (Free names of terms and channels):

$$\begin{aligned}\mathrm{fn}(n) &= \emptyset\\ \mathrm{fn}(a) &= \{a\}\\ \mathrm{fn}(x) &= \emptyset\\ \mathrm{fn}(T_1 \odot T_2) &= \mathrm{fn}(T_1) \cup \mathrm{fn}(T_2)\end{aligned}$$

Definition 1.5 (Free names of processes):

$$\begin{aligned}\mathrm{fn}(\mathbf{0}) &= \emptyset\\ \mathrm{fn}(P \mid Q) &= \mathrm{fn}(P) \cup \mathrm{fn}(Q)\\ \mathrm{fn}(!P) &= \mathrm{fn}(P)\\ \mathrm{fn}(\nu a.P) &= \mathrm{fn}(P) \setminus \{a\}\\ \mathrm{fn}\Big(\overline{c}\langle\vec{T}\rangle.P\Big) &= \mathrm{fn}(c) \cup \mathrm{fn}(T) \cup \mathrm{fn}(P)\\ \mathrm{fn}\Big(\overline{c}{:}\langle\vec{T}\rangle.P\Big) &= \mathrm{fn}(c) \cup \mathrm{fn}(T) \cup \mathrm{fn}(P)\\ \mathrm{fn}\Big(c\big(\vec{x}\big)\Big) &= \mathrm{fn}(c) \cup \mathrm{fn}(P)\\ \mathrm{fn}([T_1 \bowtie T_2]P, Q) &= \mathrm{fn}(T_1) \cup \mathrm{fn}(T_2) \cup \mathrm{fn}(P) \cup \mathrm{fn}(Q)\end{aligned}$$

Definition 1.6 (Free variables of labels):

$$\begin{aligned}\mathrm{fv}(n) &= \emptyset\\ \mathrm{fv}(x) &= \{x\}\\ \mathrm{fv}(\texttt{all}) &= \emptyset\\ \mathrm{fv}(\texttt{len}) &= \emptyset\\ \mathrm{fv}(\texttt{tup}) &= \emptyset\end{aligned}$$

Definition 1.7 (Free variables of channels):

$$\begin{aligned}\mathrm{fv}(a) &= \emptyset\\ \mathrm{fv}(x) &= \{x\}\\ \mathrm{fv}(a \cdot l) &= \mathrm{fv}(l)\\ \mathrm{fv}(x \cdot l) &= \{x\} \cup \mathrm{fv}(l)\end{aligned}$$

Definition 1.8 (Free variables of terms):

$$\begin{aligned}\mathrm{fv}(n) &= \emptyset\\ \mathrm{fv}(a) &= \emptyset\\ \mathrm{fv}(x) &= \{x\}\\ \mathrm{fv}(T_1 \odot T_2) &= \mathrm{fv}(T_1) \cup \mathrm{fv}(T_2)\end{aligned}$$

Definition 1.9 (Freee variables of processes):

$$\begin{aligned}
\mathrm{fv}(\mathbf{0}) &= \emptyset \\
\mathrm{fv}(P \mid Q) &= \mathrm{fv}(P) \cup \mathrm{fv}(Q) \\
\mathrm{fv}(!P) &= \mathrm{fv}(P) \\
\mathrm{fv}(\nu a.P) &= \mathrm{fv}(P) \\
\mathrm{fv}\Big(\overline{c}\langle\vec{T}\rangle.P\Big) &= \mathrm{fv}(c) \cup \mathrm{fv}(T) \cup \mathrm{fv}(P) \\
\mathrm{fv}\Big(\overline{c}{:}\langle\vec{T}\rangle.P\Big) &= \mathrm{fv}(c) \cup \mathrm{fv}(T) \cup \mathrm{fv}(P) \\
\mathrm{fv}\Big(c\Big(\vec{x}\Big)\Big) &= \mathrm{fv}(c) \cup \mathrm{fv}(P) \\
\mathrm{fv}([T_1 \bowtie T_2]P, Q) &= \mathrm{fv}(T_1) \cup \mathrm{fv}(T_2) \cup \mathrm{fv}(P) \cup \mathrm{fv}(Q)
\end{aligned}$$

Definition 1.10 (Bound names of processes):

$$\begin{aligned}
\mathrm{bn}(\mathbf{0}) &= \emptyset \\
\mathrm{bn}(P \mid Q) &= \mathrm{bn}(P) \cup \mathrm{bn}(Q) \\
\mathrm{bn}(!P) &= \mathrm{bn}(P) \\
\mathrm{bn}(\nu a.P) &= \{a\} \cup \mathrm{bn}(P) \\
\mathrm{bn}\Big(\overline{c}\langle\vec{T}\rangle.P\Big) &= \mathrm{bn}(P) \\
\mathrm{bn}\Big(\overline{c}{:}\langle\vec{T}\rangle.P\Big) &= \mathrm{bn}(P) \\
\mathrm{bn}\Big(c\Big(\vec{x}\Big)\Big) &= \mathrm{bn}(P) \\
\mathrm{bn}([T_1 \bowtie T_2]P, Q) &= \mathrm{bn}(P) \cup \mathrm{bn}(Q)
\end{aligned}$$

Definition 1.11 (Bound variables of processes):

$$\begin{aligned}
\mathrm{bv}(\mathbf{0}) &= \emptyset \\
\mathrm{bv}(P \mid Q) &= \mathrm{bv}(P) \cup \mathrm{bv}(Q) \\
\mathrm{bv}(!P) &= \mathrm{bv}(P) \\
\mathrm{bv}(\nu a.P) &= \mathrm{bv}(P) \\
\mathrm{bv}\Big(\overline{c}\langle\vec{T}\rangle.P\Big) &= \mathrm{bv}(P) \\
\mathrm{bv}\Big(\overline{c}{:}\langle\vec{T}\rangle.P\Big) &= \mathrm{bv}(P) \\
\mathrm{bv}\Big(c\big(\vec{x}\big)\Big) &= \vec{x} \cup \mathrm{bv}(P) \\
\mathrm{bv}([T_1 \bowtie T_2]P, Q) &= \mathrm{bv}(P) \cup \mathrm{bv}(Q)
\end{aligned}$$

Definition 1.12 (Substitution of labels):

$$\begin{aligned}
n\Big\{{}^{T_2}/_{T_1}\Big\} &= n \\
x\Big\{{}^{T_2}/_{T_1}\Big\} &= \begin{cases} T_2 & \text{if } x = T_1 \\ x & \text{otherwise} \end{cases} \\
\texttt{all}\,\Big\{{}^{T_2}/_{T_1}\Big\} &= \texttt{all} \\
\texttt{len}\,\Big\{{}^{T_2}/_{T_1}\Big\} &= \texttt{len} \\
\texttt{tup}\,\Big\{{}^{T_2}/_{T_1}\Big\} &= \texttt{tup}
\end{aligned}$$

Definition 1.13 (Substitution of channels):

$$\begin{aligned}
a\Big\{{}^{T_2}/_{T_1}\Big\} &= \begin{cases} T_2 & \text{if } a = T_1 \\ a & \text{otherwise} \end{cases} \\
x\Big\{{}^{T_2}/_{T_1}\Big\} &= \begin{cases} T_2 & \text{if } x = T_1 \\ x & \text{otherwise} \end{cases} \\
(a \cdot T)\Big\{{}^{T_2}/_{T_1}\Big\} &= \begin{cases} T_2 \cdot \Big(T\Big\{{}^{T_2}/_{T_1}\Big\}\Big) & \text{if } a = T_1 \\ a \cdot \Big(T\Big\{{}^{T_2}/_{T_1}\Big\}\Big) & \text{otherwise} \end{cases} \\
(x \cdot T')\Big\{{}^{T_2}/_{T_1}\Big\} &= \begin{cases} T_2 \cdot \Big(T\Big\{{}^{T_2}/_{T_1}\Big\}\Big) & \text{if } x = T_1 \\ x \cdot \Big(T\Big\{{}^{T_2}/_{T_1}\Big\}\Big) & \text{otherwise} \end{cases}
\end{aligned}$$

Definition 1.14 (Substitution of terms and channels):

$$\begin{aligned}
n\Big\{{}^{T_2}/_{T_1}\Big\} &= n \\
a\Big\{{}^{T_2}/_{T_1}\Big\} &= \begin{cases} T_2 & \text{if } a = T_1 \\ a & \text{otherwise} \end{cases} \\
x\Big\{{}^{T_2}/_{T_1}\Big\} &= \begin{cases} T_2 & \text{if } x = T_1 \\ x & \text{otherwise} \end{cases} \\
(T_3 \odot T_4)\Big\{{}^{T_2}/_{T_1}\Big\} &= T_3\Big\{{}^{T_2}/_{T_1}\Big\} \odot T_4\Big\{{}^{T_2}/_{T_1}\Big\}
\end{aligned}$$

Definition 1.15 (Substitution of processes):

$$
\begin{aligned}
\mathbf{0}\left\{{}^{T_2}/_{T_1}\right\} &= \mathbf{0} \\
(P \mid Q)\left\{{}^{T_2}/_{T_1}\right\} &= P\left\{{}^{T_2}/_{T_1}\right\} \mid Q\left\{{}^{T_2}/_{T_1}\right\} \\
!P\left\{{}^{T_2}/_{T_1}\right\} &= !\left(P\left\{{}^{T_2}/_{T_1}\right\}\right) \\
(\nu a.P)\left\{{}^{T_2}/_{T_1}\right\} &= \begin{cases} \nu a.\left(P\left\{{}^{T_2}/_{T_1}\right\}\right) & \text{if } a \notin \text{fn}(T_1) \cup \text{fn}(T_2) \\ \nu b.\left(P\{{}^{b}/_{a}\}\left\{{}^{T_2'}/_{T_1'}\right\}\right) & \text{otherwise where } b \notin f_n \end{cases} \\
& \quad \text{where } f_n = \text{fn}(T_1) \cup \text{fn}(T_2) \cup \text{fn}(P) \\
& \quad T_1' = T_1\{{}^{b}/_{a}\} \text{ and } T_2' = T_2\{{}^{b}/_{a}\} \\
\left(\overline{c}\langle\vec{T}\rangle.P\right)\left\{{}^{T_2}/_{T_1}\right\} &= \left(\overline{c}\left\{{}^{T_2}/_{T_1}\right\}\right)\langle\left(\vec{T}\left\{{}^{T_2}/_{T_1}\right\}\right)\rangle.\left(P\left\{{}^{T_2}/_{T_1}\right\}\right) \\
\left(\overline{c}{:}\langle\vec{T}\rangle.P\right)\left\{{}^{T_2}/_{T_1}\right\} &= \left(\overline{c}{:}\left\{{}^{T_2}/_{T_1}\right\}\right)\left(\langle\vec{T}\left\{{}^{T_2}/_{T_1}\right\}\rangle\right).\left(P\left\{{}^{T_2}/_{T_1}\right\}\right) \\
\left(c\left(\vec{x}\right).P\right)\left\{{}^{T_2}/_{T_1}\right\} &= \begin{cases} c\left\{{}^{T_2}/_{T_1}\right\}\left(\vec{x}\right).P\left\{{}^{T_2}/_{T_1}\right\} & \text{if } \vec{x} \cap \text{fv}(T_1) \cup \text{fv}(T_2) = \emptyset \\ c\left\{{}^{T_2'}/_{T_1'}\right\}\left(\vec{y}\right).P\left\{{}^{\vec{y}}/_{\vec{x}}\right\}\left\{{}^{T_2'}/_{T_1'}\right\} & \text{otherwise where } \vec{y} \cap f_v = \emptyset \end{cases} \\
& \quad \text{where } f_v = (\text{fv}(T_1) \cup \text{fv}(T_2) \cup \text{fv}(P)) \\
& \quad T_1' = T_1\left\{{}^{\vec{y}}/_{\vec{x}}\right\} \text{ and } T_2' = T_2\left\{{}^{\vec{y}}/_{\vec{x}}\right\} \\
([T_1 \bowtie T_2]P, Q)\left\{{}^{T_2}/_{T_1}\right\} &= \bowtie T_2\left\{{}^{T_2}/_{T_1}\right\}]P\left\{{}^{T_2}/_{T_1}\right\}, Q\left\{{}^{T_2}/_{T_1}\right\}
\end{aligned}
$$

B Proofs About the Type System for BtF

B.1 Proof of Lemma 3.1

Proof of Lemma 3.1.

We prove Lemma 3.1 by inspection of the type rules.

$\boldsymbol{\tau} = \mathbf{Int}$: In the case that $\tau = \mathbf{Int}$ the only rule that gives a value this type is (BT-Int)

$\boldsymbol{\tau} = \boldsymbol{\tau_1} \to \boldsymbol{\tau_2}$: In the case that $\tau_1 \to \tau_2$ the only rule that gives a value this type is (BT-Abs)

$\boldsymbol{\tau} = [\boldsymbol{\tau}]$: In the case that $\tau = [\tau]$ the only rule that gives a value this type is (BT-Array)

$\boldsymbol{\tau} = \left(\vec{\boldsymbol{\tau}}\right)$: In the case that $\tau = \left(\vec{\tau}\right)$ the only rule that gives a value this type is (BT-Tuple)

□

B.2 Proof of Lemma 3.2

Proof of Lemma 3.2.

By induction on the depth of the derivation of $\Gamma \vdash e : \tau$

(**BT-Int**): Then $e = n$ and $\tau = \mathbf{Int}$. Therefore by (BT-Int) we have $\Gamma, x : \tau' \vdash n : \mathbf{Int}$.

(**BT-Var**): Then $e = y$ and $y : \tau \in \mathrm{dom}(\Gamma)$. Since $x \notin \mathrm{dom}(\Gamma)$, we have that $x \neq y$. Therefore we get $y \in \mathrm{dom}(\Gamma, x : \tau')$ and from (BT-Var) we have $\Gamma, x : \tau' \vdash y : \tau$.

(**BT-Bin**): Then $e = e_1 \odot e_2$ and $\Gamma \vdash e_1 : \mathbf{Int}$ and $\Gamma \vdash e_2 : \mathbf{Int}$ and $\tau = \mathbf{Int}$. Therefore using the inductive hypothesis we have $\Gamma, x : \tau' \vdash e_1 : \mathbf{Int}$ and $\Gamma, x : \tau' \vdash e_2 : \mathbf{Int}$ and from (BT-Bin) we get $\Gamma, x : \tau' \vdash e_1 \odot e_2 : \tau$.

(**BT-If**): Then $e = \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$ and $\Gamma \vdash e_1 : \mathbf{Int}$ and $\Gamma \vdash e_2 : \tau$ and $\Gamma \vdash e_3 : \tau$. From the induction hypothesis we have that $\Gamma, x : \tau' \vdash e_1 : \mathbf{Int}$ and $\Gamma, x : \tau' \vdash e_2 : \tau$ and $\Gamma, x : \tau' \vdash e_2 : \tau$. Therefore, from (BT-If) we have $\Gamma, x : \tau' \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau$.

(**BT-Array**): Then $e = [e_1, ..., e_n]$ and $\forall i \in \{1, ..., n\}.\Gamma \vdash e_i = \tau_1$ and $\tau = [\tau_1]$. From the inductive hypothesis we have that $\forall i \in \{1, ..., n\}.\Gamma, x : \tau' \vdash e_i : \tau_1$ and from (BT-Array) we get $\Gamma, x : \tau' \vdash [e_1, ..., e_n] : [\tau_1]$

(**BT-Tuple**): Then $e = (e_1, ..., e_n)$ and $\forall i \in \{1, ..., n\}.\Gamma \vdash e_i = \tau_i$ and $\tau = (\tau_1, ..., \tau_n)$. From the inductive hypothesis we have that $\forall i \in \{1, ..., n\}.\Gamma, x : \tau' \vdash e_i : \tau_i$ and from (BT-Tuple) we get $\Gamma, x : \tau' \vdash (e_1, ..., e_n) : (\tau_1, ..., \tau_n)$

(**BT-Index**): Then $e = e_1[e_2]$ and $\Gamma \vdash e_1 : [\tau]$ and $\Gamma \vdash e_2 : \mathbf{Int}$. Therefore using the inductive hypothesis we have $\Gamma, x : \tau' \vdash e_1 : [\tau]$ and $\Gamma, x : \tau' \vdash e_2 : \mathbf{Int}$ and from (BT-Index) we get $\Gamma, x : \tau' \vdash e_1[e_2] : \tau$.

(**BT-Abs**): Then $e = \lambda y : \tau_1.e_1$ and $\tau = \tau_1 \to \tau_2$ and $\Gamma, y : \tau_1 \vdash e_1 : \tau_2$ we assume $x \neq y$ renaming y if needed. Because $x \notin \mathrm{dom}(\Gamma)$ it must hold that $x \notin \mathrm{dom}(\Gamma, y : \tau_1)$.

Therefore using the inductive hypothesis we get $\Gamma, y : \tau_1, x : \tau' \vdash e_1 : \tau_2$ and from (BT-Abs) we get $\Gamma, x : \tau' \vdash \lambda y : \tau_1.e_1 : \tau_1 \to \tau_2$.

(**BT-App**): Then $e = e_1 e_2$ and $\Gamma \vdash e_1 : \tau_1 \to \tau$ and $\Gamma \vdash e_2 : \tau_1$. Therefore using the inductive hypothesis we have $\Gamma, x : \tau' \vdash e_1 : \tau_1 \to \tau$ and $\Gamma, x : \tau' \vdash e_2 : \tau_1$ and from (BT-App) we get $\Gamma, x : \tau' \vdash e_1 e_2 : \tau$.

□

B.3 Proof of Lemma 3.3

Proof of Lemma 3.3.

By induction on derivations of $\Gamma, x : \tau \vdash e : \tau'$.

(**BT-Int**): Then $e = n$ and $\tau' = \mathbf{Int}$. Then from $n\{x \mapsto s\} : \tau'$ the preservation of under substitution is immediately obvious.

(**BT-Var**): Then $e = y$ and $y : \tau' \in \Gamma, (x : \tau)$. For (BT-Var) there are two sub cases depending on whether e is x or another variable .

$\boldsymbol{y = x}$: Then we get $y\{x \mapsto s\} = s$ from Definition 1.2. From the inductive hypothesis we then get that $\Gamma \vdash s : \tau$.

$\boldsymbol{y \neq x}$: Then we get $y\{x \mapsto s\} = y$ from Definition 1.2, and as the expression remains unchanged the preservation of types is immediately obvious.

(**BT-Bin**): Then $e = e_1 \odot e_2$ and $\Gamma(x : \tau) \vdash e_1 : \mathbf{Int}$ and $\Gamma(x : \tau) \vdash e_2 : \mathbf{Int}$. Using the inductive hypothesis we have that $\Gamma \vdash e_1\{x \mapsto s\} : \mathbf{Int}$ and $\Gamma \vdash e_2\{x \mapsto s\} : \mathbf{Int}$. Then using (BT-Bin) we get $\Gamma \vdash e_1\{x \mapsto s\} \odot e_2\{x \mapsto s\} : \mathbf{Int}$; therefore the type is preserved under substitution.

(**BT-If**): Then $e = \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$ and $\Gamma, (x : \tau) \vdash e_1 : \mathbf{Int}$ and $\Gamma, (x : \tau) \vdash e_2 : \tau'$ and and $\Gamma, (x : \tau) \vdash e_3 : \tau'$. Using the inductive hypothesis we have that $\Gamma \vdash e_1\{x \mapsto s\} : \mathbf{Int}$ and $\Gamma \vdash e_2\{x \mapsto s\} : \tau'$ and $\Gamma \vdash e_3\{x \mapsto s\} : \tau'$. Then using (BT-If) we get $\Gamma \vdash \texttt{if}\ e_1\{x \mapsto s\}\ \texttt{then}\ e_2\{x \mapsto s\}\ \texttt{else}\ e_3\{x \mapsto s\} : \tau'$; therefore the type is preserved under substitution.

(**BT-Array**): Then $e = [e_1, ..., e_n]$ and $\forall i \in \{1...n\}.\Gamma(x : \tau) \vdash e_i : \tau_1$ and $\tau' = [\tau_1]$. Using the inductive hypothesis we have that $\forall i \in \{1...n\}.\Gamma \vdash e_{i\{x \mapsto s\}} : \tau_1$. Then using (BT-Array) we get $\Gamma \vdash [e_1\{x \mapsto s\}, ..., e_n\{x \mapsto s\}] : [\tau_1]$; therefore the type is preserved under substitution.

(**BT-Tuple**): Then $e = (e_1, ..., e_n)$ and $\forall i \in \{1...n\}.\Gamma(x : \tau) \vdash e_i : \tau_i$ and $\tau' = (\tau_1, ..., \tau_n)$. Using the inductive hypothesis we have that $\forall i \in \{1...n\}.\Gamma \vdash e_{i\{x \mapsto s\}} : \tau_1 i$. Then using (BT-Array) we get $\Gamma \vdash (e_1\{x \mapsto s\}, ..., e_n\{x \mapsto s\}) : (\tau_1, ..., \tau_i)$; therefore the type is preserved under substitution.

(**BT-Index**): Then $e = e_1[e_2]$ and $\Gamma(x : \tau) \vdash e_1 : [\tau']$ and $\Gamma(x : \tau) \vdash e_2 : \mathbf{Int}$. Using the inductive hypothesis we have that $\Gamma \vdash e_1\{x \mapsto s\} : [\tau']$ and $\Gamma \vdash e_2\{x \mapsto s\} : \mathbf{Int}$. The

using (BT-Index) we get $\Gamma \vdash e_1\{x \mapsto s\}[e_2\{x \mapsto s\}] : \tau'$; therefore the type is preserved under substitution.

(**BT-Abs**): Then $e = \lambda(y : \tau_1).e_1$, $\tau' = \tau_1 \to \tau_2$ and $\Gamma, x : \tau, y : \tau_1 \vdash e_1 : \tau_2$ since $\Gamma \vdash s : \tau$ by Lemma 3.2 we have $\Gamma, y : \tau_1 \vdash s : \tau$, then using the inductive hypothesis we have $\Gamma, y : \tau_1 \vdash e_1\{x \mapsto s\} : \tau_2$. Therefore from (BT-Abs) we get $\Gamma \vdash \lambda(y : \tau_1).e_1\{x \mapsto s\} : \tau_1 \to \tau_2$. As we can assume $y \neq x$ from Lemma 3.2 and from Definition 1.2 we have that $e\{x \mapsto s\} = \lambda(y : \tau_1).e_1\{x \mapsto s\}$, therefore the type is preserved under substitution.

(**BT-App**): Then $e = e_1 e_2$ and $\Gamma, x : \tau' \vdash e_1 : \tau_1 \to \tau$ and $\Gamma, x : \tau' \vdash e_2 : \tau_1$. Then using the inductive hypothesis we have $\Gamma \vdash e_1\{x \mapsto s\} : \tau_1 \to \tau$ and $\Gamma \vdash e_2\{x \mapsto s\} : \tau_1$. Therefore using (BT-App) we get $\Gamma \vdash e_1\{x \mapsto s\}e_2\{x \mapsto s\} : \tau$. Because we have $e\{x \mapsto s\} = e_1\{x \mapsto s\}e_2\{x \mapsto s\}$ from Definition 1.2, therefore the type is preserved under substitution.

□

B.4 Proof of Theorem 3.1 - preservation

Proof of preservation.

Induction on the derivation of $\Gamma \vdash e : \tau$ using case analysis on the last rule in the derivation.

(**BT-Int**): Then $e = n$, and by inspection of the reduction rules there exists no e' such that $n \to e'$ making the preservation immediately obvious.

(**BT-Var**): Then $e = x$, and by inspection of the reduction rules there exists no e' such that $x \to e'$ making the preservation immediately obvious.

(**BT-Bin**): Then $e = e_1 \odot e_2$ and $\tau = \mathbf{Int}$ and $\Gamma \vdash \odot : \mathbf{Int} \to \mathbf{Int} \to \mathbf{Int}$ and $\Gamma \vdash e_1 : \mathbf{Int}$ and $\Gamma \vdash e_2 : \mathbf{Int}$. By assuming $e \to e'$ exists, we derive the following reduction rules:

(**B-Bin1**): then $e' = e_1' \odot e_2$, where $e_1 \to e_1'$. Using the induction hypothesis we get that $\Gamma \vdash e_1' : \mathbf{Int}$ therefore, using (BT-Bin) we get that $\Gamma \vdash e' : \tau$.

(**B-Bin2**): then $e' = e_1 \odot e_2'$, where $e_2 \to e_2'$. Using the induction hypothesis we get that $\Gamma \vdash e_2' : \mathbf{Int}$ therefore, using (BT-Bin) we get that $\Gamma \vdash e' : \tau$.

(**B-Bin**): then $e' = v_3$ and from the typing of $\odot$ we get that $\Gamma \vdash e' : \tau$.

(**BT-If**): Then $e = \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$ and $e_1 : \mathbf{Int}$ and $e_2 : \tau$ and $e_3 : \tau$. By assuming $e \to e'$ exists, we then derive the following applicable reduction rules:

(**B-Ift**): then $e' = e_2$ and from the typing of e_2 we get $\Gamma \vdash e' : \tau$.

(**B-Iff**): then $e' = e_3$ and from the typing of e_3 we get $\Gamma \vdash e' : \tau$.

(**B-If**): then we get $e' = \texttt{if}\ e_1'\ \texttt{then}\ e_2\ \texttt{else}\ e_3$, where $e_1 \to e_1'$. Using the inductive hypothesis we get that $\Gamma \vdash e_1' : \mathbf{Int}$, therefore using (BT-If) we get $\Gamma \vdash e' : \tau$.

(**BT-Array**): Then $e = [e_1, ..., e_n]$ and $\forall i \in \{1, ..., n\}.\Gamma \vdash e_i : \tau_1$ and $\tau = [\tau_1]$. By assuming $e \to e'$ exists, we then derive that (B-Array) is the only applicable reduction rule.

Therefore $e' = [e_1, ..., e'_i, ..., e_n]$. Using the inductive hypothesis we get that $\Gamma \vdash e'_1 : \tau_1$, using (BT-Array) we get $\Gamma \vdash e' : \tau$.

(**BT-Tuple**): Then $e = (e_1, ..., e_n)$ and $\forall i \in \{1, ..., n\}.\Gamma \vdash e_i : \tau_i$ and $\tau = (t_1, ..., t_n)$. By assuming $e \to e'$ exists, we then derive that (B-Tuple) is the only applicable reduction rule. Therefore $e' = (e_1, ..., e'_i, ..., e_n)$. Using the inductive hypothesis we get that $\Gamma \vdash e'_1 : \tau_1$, using (BT-Tuple) we get $\Gamma \vdash e' : \tau$.

(**BT-Index**): Then $e = e_1[e_2]$ and $\Gamma \vdash e_1 : [\tau]$ and $\Gamma \vdash e_2 : \mathbf{Int}$. By assuming $e \to e'$ exists, we then derive the following reduction rules:

(**B-Index1**): then $e' = e'_1[e_2]$, where $e_1 \to e'_1$. Using the inductive hypothesis we get that $\Gamma \vdash e'_1 : [\tau]$, therefore using (BT-Index) we get $\Gamma \vdash e' : \tau$.

(**B-Index2**): then $e' = e_1[e'_2]$, where $e_2 \to e'_2$. Using the inductive hypothesis we get that $\Gamma \vdash e'_2 : \mathbf{Int}$, therefore using (BT-Index) we get $\Gamma \vdash e' : \tau$.

(**B-Index**): then $e' = v_i$, using Lemma 3.1 we get that $e_1 = [v_1, ..., v_n]$. Then using (BT-Array) we get that $\forall i \in \{1, ..., n\}.\Gamma \vdash v_i : \tau$ and therefore we get that $\Gamma \vdash v_i : \tau$.

(**BT-Abs**): Then $e = \lambda(x : \tau).e$, and by inspection of the reduction rules there exists no e' such that $\lambda(x : \tau).e \to e'$ making the preservation immediately obvious.

(**BT-App**): Then $e = e_1 e_2$, and $\Gamma \vdash e_1 : \tau_1 \to \tau$ and $\Gamma \vdash e_2 : \tau_1$. By assuming $e \to e'$ exists, and using case analysis we derive the following applicable reduction rules:

(**B-App1**): Then $e' = e'_1 e_2$ and $e_1 \to e'_1$. Using the induction hypothesis we have that $\Gamma \vdash e'_1 : \tau_1 \to \tau$. Therefore, using (BT-App) we get $\Gamma \vdash e'_1 e_2 : \tau$.

(**B-App2**): Then $e' = e_1 e'_2$ and $e_2 \to e'_2$. Using the induction hypothesis we have that $\Gamma \vdash e'_2 : \tau_1$. Therefore, using (BT-App) we get $\Gamma \vdash e_1 e'_2 : \tau$.

(**B-Abs**): Then $e_1 = \lambda(x : \tau_2).e_3$ and $e_2 = v$ and $e' = e_3\{x \mapsto v\}$. Because $\Gamma \vdash e_1 : \tau_1 \to \tau$ and $e_1 = \lambda(x : \tau_2).e_3$ by inspection of the type rules it must hold that $\tau_1 = \tau_2$ giving us $\Gamma \vdash \lambda(x : \tau_1).e_3 : \tau_1 \to \tau$. Then by inspection the derivation must end with (BT-Abs) giving us $\Gamma, (x : \tau_1)e_3 : \tau$. Then because $\Gamma \vdash e_2 : \tau_1$ and $e_2 = v$ it must hold that $\Gamma \vdash v : \tau_1$. Therefore, using the Lemma 3.3 we have that $\Gamma \vdash e_3\{x \mapsto v\} : \tau$.

(**B-Map**): Then $e_1 = \texttt{map}$ for e_2 there are two cases that apply:

$\boldsymbol{e_2 \notin \mathcal{V}}$: the case for (B-App2) applies.

$\boldsymbol{e_2 \in \mathcal{V}}$: using Lemma 3.1 we get that $e_2 = (\lambda(x : \tau_1) : e_3, [v_1, ..., v_n])$ then we get that $e' = [e_3\{x \mapsto v_1\}, ..., e_3\{x \mapsto v_n\}]$. Then from Definition 3.3 we have that $\Gamma \vdash \texttt{map} : (\mathbf{Int} \to \mathbf{Int}, [\mathbf{Int}]) \to [\mathbf{Int}]$ then by inspection of the type rules we have that $\tau = [\mathbf{Int}]$ and $\tau_1 = \mathbf{Int}$. Then by inspection the derivation must end with (BT-Array) giving us $\Gamma \vdash [v_1, ..., v_n] : \tau$ and $\forall i \in \{1, ..., n\}.\Gamma \vdash v_i : \mathbf{Int}$. Therefore using Lemma 3.3 we have that $\Gamma \vdash [e\{x \mapsto v_1\}, ..., e\{x \mapsto v_n\}] : \tau$.

(**B-Iota**): Then $e_1 = \texttt{iota}$ for e_2 there are two cases that apply:

$\boldsymbol{e_2} \notin \boldsymbol{\mathcal{V}}$: the case for (B-App2) applies.

$\boldsymbol{e_2} \in \boldsymbol{\mathcal{V}}$: using Lemma 3.1 we get that $e_2 = n$, then we get that $e' = [v_1, ..., v_n]$. Then from Definition 3.3 we have that $\Gamma \vdash \texttt{iota} : \textbf{Int} \to [\textbf{Int}]$ then by inspection of the type rules we have that $\tau = [\textbf{Int}]$ and $\tau_1 = \textbf{Int}$. Then using (BT-Array) we get that $\Gamma \vdash e' : \tau$.

(**B-Size**): Then $e_1 = \texttt{size}$ for e_2 there are two cases that apply:

$\boldsymbol{e_2} \notin \boldsymbol{\mathcal{V}}$: the case for (B-App2) applies.

$\boldsymbol{e_2} \in \boldsymbol{\mathcal{V}}$: using Lemma 3.1 we get that $e_2 = [v_1, ..., v_n]$ then we get that $e' = n$. Then from Definition 3.3 we have that $\Gamma \vdash \texttt{size} : [\textbf{Int}] \to \textbf{Int}$ then by inspection of the type rules we have that $\tau = \textbf{Int}$ and $\tau_1 = [\textbf{Int}]$. Then using (BT-Int) we get that $\Gamma \vdash e' : \tau$.

□

B.5 Proof of Theorem 3.1 - progress

Proof of progress.

We will prove progress by induction on the typing derivation of $\vdash e : \tau$ (the empty environment). We will go over each type rule and show that progress holds: either e is a value or it can take a step $e \to e'$.

(**BT-Int**): Trivial as n is a value and therefore progress holds.

(**BT-Var**): We cannot type (BT-Var) under the empty environment. We would have that $x : \tau$ under Γ but that contradicts our inductive hypothesis.

(**BT-Abs**): The case of (BT-Abs) is trivial as abstraction is a value and therefore progress holds.

(**BT-App**): We know by (BT-App) that e_1 has type $\tau_1 \to \tau_2$. By the induction hypothesis we know that $e_1 \in \mathcal{V}$ or $\exists e_1'.e_1 \to e_1'$ and $e_2 \in \mathcal{V}$ or $\exists e_2'.e_2 \to e_2'$. That gives us four cases:

$\boldsymbol{e_1, e_2} \notin \boldsymbol{\mathcal{V}}$: In the case that e_1 and e_2 are not values then by our inductive hypothesis $e_1 \to e_1'$ and $e_2 \to e_2'$. If e_1 takes a step then (B-App1) applies. From (B-App1) we have that given the premise we can take the step $e_1 e_2 \to e_1' e_2$ and therefore progress holds.

if e_2 takes a step then (B-App2) applies. From (B-App2) we have that given the premise we can take the step $e_1 e_2 \to e_1 e_2'$ and therefore progress holds.

$\boldsymbol{e_1} \notin \boldsymbol{\mathcal{V}}$: In the case that e_1 is not a value then by our inductive hypothesis it can take a step. Then the rule (B-App1) applies. From (B-App1) we have that given the premise we can take the step $e_1 e_2 \to e_1' e_2$ and therefore progress holds.

$\boldsymbol{e_2} \notin \boldsymbol{\mathcal{V}}$: In the case that e_2 is not a value then by our inductive hypothesis it can take a step. Then the rule (B-App2) applies. From (B-App2) we have that given the premise we can take the step $e_1 e_2 \to e_1 e_2'$ and therefore progress holds.

$\boldsymbol{e_1}, \boldsymbol{e_2} \in \boldsymbol{\mathcal{V}}$: In the last case both e_1 and e_2 are values. As e_1 must be an arrow type (by our type rule) - then abstraction applies (as that is the only value that has type $\tau_1 \to \tau_2$ from proof of Lemma 3.1) and we can use (B-Abs) to take a step.

(**BT-Bin**): We know from (BT-Bin) that e_1 and e_2 has type **Int**. By the induction hypothesis we know that $e_1 \in \mathcal{V}$ or $\exists e_1'.e_1 \to e_1'$ and $e_2 \in \mathcal{V}$ or $\exists e_2'.e_2 \to e_2'$. This gives us the following cases:

$\boldsymbol{e_1}, \boldsymbol{e_2} \notin \boldsymbol{\mathcal{V}}$: In the case that $e_1, e_2 \notin \mathcal{V}$ the by our inductive hypothesis we know that $e_1 \to e_1'$ and $e_2 \to e_2'$. If e_1 take a step then (B-Bin1) applies. From (B-Bin1) we have that given the premise we can take the step $e_1 \odot e_2 \to e_1' \odot e_2$ and therefore progress applies.

If e_2 take a step then (B-Bin2) applies. From (B-Bin2) we have that given the premise we can take the step $e_1 \odot e_2 \to e_1 \odot e_2'$ and therefore progress applies.

$\boldsymbol{e_1} \notin \boldsymbol{\mathcal{V}}$: In the case that $e_1 \notin \mathcal{V}$ then by our inductive hypothesis e_1 can take a step and then (B-Bin1) applies. From (B-Bin1) we have that given the premise we can take the step $e_1 \odot e_2 \to e_1' \odot e_2$ and therefore progress applies.

$\boldsymbol{e_2} \notin \boldsymbol{\mathcal{V}}$: In the case that $e_2 \notin \mathcal{V}$ then by our inductive hypothesis e_2 can take a step and then (B-Bin2) applies. From (B-Bin2) we have that given the premise we can take the step $e_1 \odot e_2 \to e_1 \odot e_2'$ and therefore progress applies.

$\boldsymbol{e_1}, \boldsymbol{e_2} \in \boldsymbol{\mathcal{V}}$: In the last case both e_1 and e_2 are values. Both values must be numbers (that is the only value of type **Int** by proof of Lemma 3.1). In that case (B-Bin) applies and given the premise e can take a step and therefore progress applies.

(**BT-If**): We know from (BT-If) that e_1 has type **Int**. By the inductive hypothesis we know that either $e_1 \in \mathcal{V}$ or $\exists e_1'.e_1 \to e_1'$. This gives us two cases:

$\boldsymbol{e_1} \notin \boldsymbol{\mathcal{V}}$: In the case that $e_1 \notin \mathcal{V}$ then by our inductive hypothesis $e_1 \to e_1'$. We can then apply (B-If). We can see that given the premise we can take the step `if` e_1 `then` e_2 `else` $e_3 \to$ `if` e_1' `then` e_2 `else` e_3 and therefore progress holds.

$\boldsymbol{e_1} \in \boldsymbol{\mathcal{V}}$: In the case that $e_1 \in \mathcal{V}$ then it must be number (that is the only value of type **Int** by proof of Lemma 3.1). In that case one of the two following rules applies: (B-Ift) or (B-Iff).

First case we apply (B-Ift) where given the premise e can take the step `if` e_1 `then` e_2 `else` $e_3 \to e_2$ and therefore progress holds.

Second case we apply (B-Iff) where given the premise e can take the step `if` e_1 `then` e_2 `else` $e_3 \to e_3$ and therefore progress holds.

(**BT-Array**): In the case of (BT-Array) we have n number of expressions. Then by our inductive hypothesis, for $i \in \{1, ..., n\}$, we know that $e_i \in \mathcal{V}$ or $\exists e_i'.e_i \to e_i'$. We will take a look at two cases:

$\exists \boldsymbol{i}.\boldsymbol{e_i} \notin \boldsymbol{\mathcal{V}}$: In the case that $\exists i.e_i \notin \mathcal{V}$ we know that from our inductive hypothesis $e_i \to e_i'$. In that case we can apply the rule (B-Array). from (B-Array) we have that

given the premise we take the step $[e_1, ..., e_i, ..., e_n] \rightarrow [e_1, ..., e'_i, ...e_n]$ and therefore progress holds.

$\forall \boldsymbol{i.e_i} \in \boldsymbol{\mathcal{V}}$: In the case that $\forall i.e_i \in \mathcal{V}$ then all sub-expressions are values. Then progress holds as arrays where all sub-expressions are values is in $\mathcal{V}$

(**BT-Tuple**): The case of (BT-Tuple) is similar to (BT-Array) as we have n number of expressions. Then by our inductive hypothesis, for $i \in \{1, ..., n\}$, we know that $e_i \in \mathcal{V}$ or $\exists e'_i.e_i \rightarrow e'_i$. We will take a look at two cases:

$\exists \boldsymbol{i.e_i} \notin \boldsymbol{\mathcal{V}}$: In the case that $\exists i.e_i \notin \mathcal{V}$ then by our inductive hypothesis there must exist an e'_i such that $e_i \rightarrow e'_i$. We can then apply the rule (B-Tuple). From (B-Tuple) we have that if the premise holds we can take the step $(e_1, ..., e_i, ..., e_n) \rightarrow (e_1, ..., e'_i, ...e_n)$ and therefore progress holds.

$\forall \boldsymbol{i.e_i} \in \boldsymbol{\mathcal{V}}$: In the case that $\forall i.e_i \in \mathcal{V}$ then all sub expressions are values. Then progress holds as tuples where all sub-expressions are values is in $\mathcal{V}$.

(**BT-Index**): We know from (BT-Index) that e_1 has type $[\tau]$ and e_2 has type **Int**. By our inductive hypothesis we know that $e_1 \in \mathcal{V}$ or $\exists e'_1.e_1 \rightarrow e'_1$ and $e_2 \in \mathcal{V}$ or $\exists e'_2.e_2 \rightarrow e'_2$. That gives us four cases:

$\boldsymbol{e_1, e_2} \notin \boldsymbol{\mathcal{V}}$: In the case that $e_1, e_2 \notin \mathcal{V}$ then by our inductive hypothesis we know that e_1 and e_2 can take a step. If e_1 takes a step then (B-Index1) applies. From (B-Index1) we have that given the premise we can take the step $e_1[e_2] \rightarrow e'_1[e_2]$ and therefore progress holds.

If e_2 take a step then (B-Index2) applies. From (B-Index2) we have that given the premise we can take the step $e_1[e_2] \rightarrow e_1[e'_2]$ and therefore progress holds.

$\boldsymbol{e_1} \notin \boldsymbol{\mathcal{V}}$: In the case that $e_1 \notin \mathcal{V}$ then by our inductive hypothesis e_1 can take a step and then (B-Index1) applies. From (B-Index1) we have that given the premise we can take the step $e_1[e_2] \rightarrow e'_1[e_2]$ and therefore progress holds.

$\boldsymbol{e_2} \notin \boldsymbol{\mathcal{V}}$: In the case that $e_2 \notin \mathcal{V}$ then by our inductive hypothesis e_2 can take a step and then (B-Index2) applies. From (B-Index2) we have that given the premise we can take the step $e_1[e_2] \rightarrow e_1[e'_2]$ and therefore progress holds.

$\boldsymbol{e_1, e_2} \in \boldsymbol{\mathcal{V}}$: In the last case both e_1 and e_2 are values. e_1 must be an array (that is the only value of type $[\tau]$ by proof of Lemma 3.1) and e_2 must be a number (that is the only value of type **Int** by proof of Lemma 3.1). In that case (B-Index) applies. We can see that given the premise then e can take a step and therefore progress holds.

□

C The *error* predicate of TEπ

$$\text{(ER-Send)}\ \frac{\Delta(c) = \mathrm{ch}\left(\vec{t_1}\right) \quad \Delta, \Pi \vdash \vec{T} : \vec{t_2} \quad \vec{t_1} \neq \vec{t_2}}{\Delta, \Pi \vdash \overline{c}\langle\vec{T}\rangle.P \xrightarrow{\tau} \mathit{error}}$$

$$\text{(ER-Broad)}\ \frac{\Delta(c) = \mathrm{ch}\left(\vec{t_1}\right) \quad \Delta, \Pi \vdash \vec{T} : \vec{t_2} \quad \vec{t_1} \neq \vec{t_2}}{\Delta, \Pi \vdash \overline{c}{:}\langle\vec{T}\rangle.P \xrightarrow{:c} \mathit{error}}$$

$$\text{(ER-Recv)}\ \frac{\Delta(c) = \mathrm{ch}\left(\vec{t_1}\right) \quad \vec{t_1} \neq \vec{t_2}}{\Delta, \Pi \vdash c\left(\vec{x} : \vec{t_2}\right).P \xrightarrow{\tau} \mathit{error}}$$

$$\text{(ER-Par1)}\ \frac{\Delta, \Pi \vdash P \xrightarrow{\tau} \mathit{error}}{\Delta, \Pi \vdash P \mid Q \xrightarrow{\tau} \mathit{error}}$$

$$\text{(ER-Par2)}\ \frac{\Delta, \Pi \vdash Q \xrightarrow{\tau} \mathit{error}}{\Delta, \Pi \vdash P \mid Q \xrightarrow{\tau} \mathit{error}}$$

$$\text{(ER-Rep)}\ \frac{\Delta, \Pi \vdash P \xrightarrow{\tau} \mathit{error}}{\Delta, \Pi \vdash !P \xrightarrow{\tau} \mathit{error}}$$

$$\text{(ER-Res)}\ \frac{P \xrightarrow{q} \mathit{error}\ \text{(temp)}}{\Delta, \Pi \vdash (\nu a : t).P \xrightarrow{q} \mathit{error}}$$

$$\text{(ER-Match)}\ \frac{\Delta, \Pi \vdash T_1 : t_1 \quad \Delta, \Pi \vdash T_2 : t_2 \quad t_1 \neq t_2}{\Delta, \Pi \vdash [T_1 \bowtie T_2]P, Q \xrightarrow{\tau} \mathit{error}}$$

$$\text{(ER-Bin1)}\ \frac{\Delta, \Pi \vdash T_1 : t \quad t \neq \mathbf{Int}}{\Delta, \Pi \vdash T_1 \odot T_2 \xrightarrow{\tau} \mathit{error}}$$

$$\text{(ER-Bin2)}\ \frac{\Delta, \Pi \vdash T_2 : t \quad t \neq \mathbf{Int}}{\Delta, \Pi \vdash T_1 \odot T_2 \xrightarrow{\tau} \mathit{error}}$$

$$\text{(ER-Compx1)}\ \frac{\Delta(u) \neq @\ell}{\Delta, \Pi \vdash u \cdot x \xrightarrow{\tau} \mathit{error}}$$

$$\text{(ER-Compn1)}\ \frac{\Delta(u) \neq @\ell}{\Delta, \Pi \vdash u \cdot n \xrightarrow{\tau} \mathit{error}}$$

$$\text{(ER-Compx2)}\ \frac{\Delta(u) = @\ell \quad \Pi(u, x) \neq \mathrm{pch}\left(\vec{t}\right)}{\Delta, \Pi \vdash u \cdot x \xrightarrow{\tau} \mathit{error}}$$

$$\text{(ER-Compn2)}\ \frac{\Delta(u) = @\ell \quad \Pi(u, n) \neq \mathrm{pch}\left(\vec{t}\right)}{\Delta, \Pi \vdash u \cdot x \xrightarrow{\tau} \mathit{error}}$$

$$\text{(ER-Compx3)}\ \frac{\Delta(u) = @\ell \quad \Pi(u, x) = \mathrm{pch}\left(\vec{t}\right) \quad \mathrm{pch}\left(\vec{t}\right) \notin \ell}{\Delta, \Pi \vdash u \cdot x \xrightarrow{\tau} \mathit{error}}$$

$$\text{(ER-Compn3)}\ \frac{\Delta(u) = @\ell \quad \Pi(u, n) = \mathrm{pch}\left(\vec{t}\right) \quad \mathrm{pch}\left(\vec{t}\right) \notin \ell}{\Delta, \Pi \vdash u \cdot n \xrightarrow{\tau} \mathit{error}}$$

Figure 3.1: Wrong processes for TEπ

D Proofs About the Type System for TEπ

D.1 Proof of Lemma 3.4

Proof of Lemma 3.4.

n: Then $T = n$, and from (ET-N) we get that $\Delta, \Pi \vdash T : \mathbf{Int}$. Then from Definition 1.4 and Definition 1.8 we get that $\mathrm{fn}(T) \cup \mathrm{fv}(T) = \emptyset$. Therefore adding $u : t_u$ to Δ and applying (ET-N) preserves the typing $\Delta, u : t_u, \Pi \vdash T : \mathbf{Int}$.

u: Then $T = u'$, and from (ET-U) we get that $\Delta, \Pi \vdash T : t$ and $\Delta(u') = t$. Since $u \notin \mathrm{dom}(\Delta)$, then $(\Delta, u : t_u, \Pi)(u') = t$ therefore applying (ET-U) preserves the typing $\Delta, u : t_u, \Pi \vdash T : t$.

***T*$_1$ $\odot$ *T*$_2$**: Then $T = T_1 \odot T_2$, and from (ET-Bin) we get that $\Delta, \Pi \vdash T : \mathbf{Int}$, $\Delta, \Pi \vdash T_1 : \mathbf{Int}$ and $\Delta, \Pi \vdash T_2 : \mathbf{Int}$. Then from the induction hypothesis we get that $\Delta, u : t_u, \Pi \vdash T_1 : \mathbf{Int}$ and $\Delta, u : t_u, \Delta, \Pi \vdash T_2 : \mathbf{Int}$. Therefore applying (ET-U) preserves the typing $\Delta, u : t_u, \Pi \vdash T : \mathbf{Int}$.

c* $\cdot$ *T: Then $T = c \cdot T'$ and as T' can be typed with either (ET-Compx) and (ET-Compn) depending on wether T' is x or n. Therefore we get that $\Delta(c) = @\ell$, $\Pi(c, T') = \mathrm{pch}(t)$, $\mathrm{pch}(t) \in @\ell$ and $\Delta, \Pi \vdash c \cdot T' : \mathrm{ch}(t)$. Therefore applying (ET-Compx) or (ET-Compn) preserves the typing $\Delta, u : t_u, \Pi \vdash c \cdot T' : \mathrm{ch}(t)$. Then as u is not in the environment by Definition 3.8 and by Definition 3.6 we know that the type of T' depends on the handle c then $c \neq u$ and $T' \neq u$. Therefore applying (ET-Compx) or (ET-Compn) with u in the environment preserves the typing $\Delta, u : t_u, \Pi \vdash T : \mathrm{ch}(t)$.

□

D.2 Proof of Lemma 3.5

Proof of Lemma 3.5.

By induction on derivation of $\Delta, \Pi \vdash P$

0: Then $P = \mathbf{0}$ and by (ET-Nil) we get that $\Delta, \Pi \vdash \mathbf{0}$ holds for any Δ, Π as $\mathbf{0}$ is not dependent on the environment.

P* | *Q: Then $P = Q \mid R$ and by (ET-Par) we get that if $\Delta, \Pi \vdash P$, then $\Delta, \Pi \vdash Q$ and $\Delta, \Pi \vdash R$. Then using the induction hypothesis we get that $\Delta, u : t_u, \Pi \vdash R$ and $\Delta, u : t_u, \Pi \vdash Q$. Therefore applying (ET-Par) gives us $\Delta, u : t_u, \Pi \vdash Q \mid R$ preserving that P is well-typed.

!*P*: Then $P = !Q$ and by (ET-Rep) we get that if $\Delta, \Pi \vdash !Q$, then $\Delta, \Pi \vdash Q$. Then using the induction hypothesis we get that $\Delta, u : t_u, \Pi \vdash Q$. Therefore applying (ET-Rep) gives us $\Delta, u : t_u, \Pi \vdash !Q$ preserving that P is well-typed.

(ν*a* : *t*).*P*: Then $P = (\nu a : t).Q$ and by (ET-Res) we get that if $\Delta, \Pi \vdash P$, then $\Delta, a : t, \Pi \vdash Q$. Then using the induction hypothesis we get that $\Delta, a : t, u : t_u, \Pi \vdash Q$. Therefore applying (ET-Res) gives us $\Delta, u : t_u, \Pi \vdash (\nu a : t).Q$ preserving that P is well-typed.

$\overline{\boldsymbol{c}}\langle\vec{\boldsymbol{T}}\rangle.\boldsymbol{P}$: Then $P = \overline{c}\langle\vec{T}\rangle.Q$ and by (ET-Send) we get that if $\Delta, \Pi \vdash P$, then $\Delta, \Pi \vdash c : \mathrm{ch}\left(\vec{t}\right)$, $\Delta, \Pi \vdash \vec{T} : \vec{t}$ and $\Delta, \Pi \vdash Q$. Then using the inductive hypothesis we get that $\Delta, u : t_u, \Pi \vdash Q$. Then by Lemma 3.4 we get that we can weaken the terms and the channel s.t $\Delta, u : t_u, \Pi \vdash c : \mathrm{ch}\left(\vec{t}\right)$ and $\Delta, u : t_u, \Pi \vdash \vec{T} : \vec{t}$. Therefore applying (ET-Send) gives us $\Delta, \Pi, u : t_u \vdash \overline{c}\langle\vec{T}\rangle.Q$ preserving that P is well-typed..

$\boldsymbol{c}\left(\vec{\boldsymbol{x}} : \vec{\boldsymbol{t}}\right).\boldsymbol{Q}$: Then $P = c\left(\vec{x}, \vec{t}\right).Q$ and by (ET-Recv) we get that if $\Delta, \Pi \vdash P$, then $\Delta, \Pi \vdash c : \mathrm{ch}\left(\vec{t}\right)$ and $\Delta, \vec{x} : \vec{t}, \Pi \vdash Q$. From Definition 3.8 we can derive that $u \notin \mathrm{fn}(P) \cup \mathrm{fv}(P)$. Since Δ, Π only contains free variables/names therefore u is a new free variable or name. Then by the induction hypothesis we get that $\Delta' = \Delta, \vec{x} : \vec{t}, u : t_u$, $\Delta', \Pi \vdash Q$. Therefore applying (ET-Recv) gives us $\Delta, u : t, \Pi \vdash c\left(\vec{x}, \vec{t}\right).Q$ preserving that P is well-typed..

$\overline{\boldsymbol{c}}{:}\langle\vec{\boldsymbol{T}}\rangle.\boldsymbol{P}$: Then $P = \overline{c}{:}\langle\vec{T}\rangle.Q$ and by (ET-Broad) we get that if $\Delta, \Pi \vdash P$, then $\Delta, \Pi \vdash c : \mathrm{ch}\left(\vec{t}\right)$, $\Delta, \Pi \vdash \vec{T} : \vec{t}$ and $\Delta, \Pi \vdash Q$. Then using the inductive hypothesis we get that $\Delta, \Pi, u : t_u \vdash Q$. Then by Lemma 3.4 we get that we can weaken the terms and the channel s.t $\Delta, \Pi, u : t_u \vdash c : \mathrm{ch}\left(\vec{t}\right)$ and $\Delta, u : t_u, \Pi \vdash \vec{T} : \vec{t}$. Therefore applying (ET-Send) gives us $\Delta, u : t_u, \Pi \vdash \overline{c}{:}\langle\vec{T}\rangle.Q$ preserving that P is well-typed.

$[\boldsymbol{T_1} \bowtie \boldsymbol{T_2}]\boldsymbol{P}, \boldsymbol{Q}$: Then $P = [T_1 \bowtie T_2]Q, R$ and by (ET-Match) we get that if $\Delta, \Pi \vdash P$, then $\Delta, \Pi \vdash T_1 : t$,$\Delta, \Pi \vdash T_2 : t$, $\Delta, \Pi \vdash Q$ and $\Delta, \Pi \vdash R$. Then using the inductive hypothesis we get that $\Delta, u : t, \Pi \vdash Q$ and $\Delta, \Pi, u : t \vdash R$. Then using Lemma 3.4 we get that we can weaken the terms s.t $\Delta, u : t, \Pi \vdash T_1 : t$, $\Delta, u : t, \Pi \vdash T_2 : t$ Therefore applying (ET-Match) we get that $\Delta, u : t, \Pi \vdash [T_1 \bowtie T_2]Q, R$ preserving that P is well-typed.

□

D.3 Proof of Lemma 3.6

Proof of Lemma 3.6.

$\boldsymbol{n}$: Then $T = n$ and from (ET-N) we get that $\Delta, u : t_u, \Pi \vdash T : \mathbf{Int}$. Then from Definition 1.4 and Definition 1.8 we get that $\mathrm{fn}(T) \cup \mathrm{fv}(T) = \emptyset$. From Definition 3.8 we can derive $u \notin \mathrm{fn}(P) \cup \mathrm{fv}(P)$. Therefore applying (ET-N) without u in the environment preserves the typing $\Delta, \Pi \vdash T : \mathbf{Int}$.

$\boldsymbol{u}$: Then $T = u'$ and from (ET-U) we get that $\Delta, u : t_u, \Pi \vdash T : t$ and $(\Delta, u : t_u, \Pi)(u') = t$.

- $\boldsymbol{u' \neq u}$: Then as $u \notin \mathrm{dom}(\Delta)$ and $u' \in \mathrm{dom}(\Delta)$ we get that $\Delta(T) : t$. Therefore applying (ET-U) without u in the environment preserves the typing $\Delta, \Pi \vdash T : t$.
- $\boldsymbol{u' = u}$: contradicts $u \notin \mathrm{dom}(\Delta)$.

$\boldsymbol{T_1 \odot T_2}$: Then $T = T_1 \cup T_2$ and from (ET-Bin) we get that $\Delta, u : t_u, \Pi \vdash T : \mathbf{Int}$, $\Delta, u : t_u, \Pi \vdash T_1 : \mathbf{Int}$ and $\Delta, u : t_u, \Pi \vdash T_2 : \mathbf{Int}$. Then using the inductive hypothesis we get that $\Delta, \Pi \vdash T_1 : \mathbf{Int}$ and $\Delta, \Pi \vdash T_2 : \mathbf{Int}$. Therefore applying (ET-Bin) preserves the typing $\Delta, \Pi \vdash T : \mathbf{Int}$.

$\boldsymbol{c \cdot T}$: Then $T = c \cdot T'$ and as T' can be typed with either (ET-Compx) and (ET-Compn) depending on wether T' is x or n. Therefore we get that $(\Delta, u : t_u)(c) = @\ell$, $\Pi(c, T') = \text{pch}(t)$, $\text{pch}(t) \in @\ell$ and $\Delta, u : t_u, \Pi \vdash c \cdot T' : \text{ch}(t)$. Then as u is not in the environment by Definition 3.8 and by Definition 3.6 we know that the type of T' depends on the handle c then $c \neq u$ and $T' \neq u$. Therefore applying (ET-Compx) or (ET-Compn) preserves the typing $\Delta, \Pi \vdash T : \text{ch}(t)$.

□

D.4 Proof of Lemma 3.7

Proof of Lemma 3.7.

0: Using (ET-Nil) we get that $\Delta, u : t, \Pi \vdash \mathbf{0}$ holds for any Δ, Π as $\mathbf{0}$ is not dependent on the environment.

$\boldsymbol{P \mid Q}$: Then $P = Q \mid R$ and by (ET-Par) we get that if $\Delta, u : t_u, \Pi \vdash P$, then $\Delta, u : t_u, \Pi \vdash Q$ and $\Delta, u : t_u, \Pi \vdash R$. Then from the induction hypothesis we have that $\Delta, u : t_u, \Pi \vdash Q$ and $\Delta, u : t_u, \Pi \vdash R$. Therefore applying (ET-Par) gives us $\Delta, \Pi \vdash Q \mid R$ preserving that P is well-typed.

$\boldsymbol{!P}$: Then $P = !Q$ and by (ET-Rep) we get that if $\Delta, u : t_u, \Pi \vdash P$, then $\Delta, u : t_u, \Pi \vdash Q$. Then from the inductive hypothesis we have that $\Delta, \Pi \vdash P$. Therefore applying (ET-Rep) gives us $\Delta, \Pi \vdash !Q$ preserving that P is well-typed.

$\boldsymbol{(\nu a : t).P}$: Then $P = (\nu a : t).Q$ and by (ET-Res) we get that if $\Delta, u : t_u, \Pi \vdash P$, then $\Delta, u : t_u, a : t, \Pi \vdash Q$. From the induction hypothesis we get that $\Delta, a : t, \Pi \vdash Q$ and that $u \notin \text{dom}(\Delta, a : t)$ therefore $u \neq a$. Therefore applying (ET-Res) gives us $\Delta, u : t_u, \Pi \vdash (\nu a : t).Q$ preserving that P is well-typed.

$\boldsymbol{\overline{c}\langle \vec{T} \rangle.P}$: Then $P = \overline{c}\langle \vec{T}.Q \rangle$ and by (ET-Send) we get that if $\Delta, u : t_u, \Pi \vdash P$, then $\Delta, u : t_u, \Pi \vdash c : \text{ch}\left(\vec{t}\right)$, $\Delta, u : t_u, \Pi \vdash \vec{T} : \vec{t}$ and $\Delta, u : t_u, \Pi \vdash Q$. Then from the inductive hypothesis we have that $\Delta, \Pi \vdash Q$. Then using Lemma 3.6 we can strengthen the terms and channel s.t $\Delta, \Pi \vdash c : \text{ch}\left(\vec{t}\right)$, $\Delta, \Pi \vdash \vec{T} : \vec{t}$. Therefore applying (ET-Send) gives us $\Delta, \Pi \vdash \overline{c}\langle \vec{T} \rangle.Q$ preserving that P is well-typed.

$\boldsymbol{c\left(\vec{x} : \vec{t}\right).Q}$: Then $P = c\left(\vec{x} : \vec{t}\right).Q$ by and (ET-Recv) we get that if $\Delta, u : t_u, \Pi \vdash P$, then $\Delta, u : t_u, \Pi \vdash c : \text{ch}\left(\vec{t}\right)$ and $\Delta, u : t_u, \vec{x} : \vec{t}, \Pi \vdash Q$. Then from the inductive hypothesis we get that $\Delta, \vec{x} : \vec{t}, \Pi \vdash Q$ and that $u \notin \text{dom}(\Delta, a : t)$ therefore $u \neq a$. Then using Lemma 3.4 we can strengthen the channel s.t $\Delta, \Pi \vdash c : \text{ch}\left(\vec{t}\right)$. Therefore applying (ET-Recv) gives us $\Delta, \Pi \vdash c\left(\vec{x} : \vec{t}\right).Q$ preserving that P is well-typed.

$\overline{\boldsymbol{c}}:\langle\vec{\boldsymbol{T}}\rangle.\boldsymbol{P}$: Then $P = \overline{c}:\langle\vec{T}.Q\rangle$ and by (ET-Broad) we get that if $\Delta, u : t_u, \Pi \vdash P$, then $\Delta, u : t_u, \Pi \vdash c : \mathrm{ch}\left(\vec{t}\right)$, $\Delta, u : t_u, \Pi \vdash \vec{T} : \vec{t}$ and $\Delta, u : t_u, \Pi \vdash Q$. Then from the inductive hypothesis we have that $\Delta, \Pi \vdash Q$. Then from Lemma 3.6 we can strengthen the terms and channel s.t $\Delta, \Pi \vdash c : \mathrm{ch}\left(\vec{t}\right)$, $\Delta, \Pi \vdash \vec{T} : \vec{t}$. Therefore applying (ET-Broad) gives us $\Delta, \Pi \vdash \overline{c}:\langle\vec{T}\rangle.Q$ preserving that P is well-typed.

$[\boldsymbol{T_1} \bowtie \boldsymbol{T_2}]\boldsymbol{P}, \boldsymbol{Q}$: Then $P = [T_1 \bowtie T_2]Q, R$ and by (ET-Match) we get that if $\Delta, u : t_u, \Pi \vdash P$, then $\Delta, u : t_u, \Pi \vdash T_1 : t$, $\Delta, u : t_u, \Pi \vdash T_2 : t$, $\Delta, u : t_u, \Pi \vdash Q$ and $\Delta, u : t_u, \Pi \vdash R$. Then from the inductive hypothesis we have that $\Delta, \Pi \vdash Q$ and $\Delta, \Pi \vdash R$. Then from Lemma 3.6 we can strengthen the terms s.t $\Delta, \Pi \vdash T_1 : t$ and $\Delta, \Pi \vdash T_2 : t$. Therefore applying (ET-Match) gives us $\Delta, \Pi \vdash [T_1 \bowtie T_2]Q, R$ preserving that P is well-typed.

□

D.5 Proof of Lemma 3.8

Proof of Lemma 3.8.

We prove Lemma 3.8 by induction in the structural congruence rules.

(**Rename**): Trivial as by α-conversion we only change bound names and in Δ, Π we only find free names, variables and numbers.

(**Par-0**): $\underbrace{P}_{1} \equiv \underbrace{P \mid \mathbf{0}}_{2}$

1: By our inductive hypothesis we have that $\Delta, \Pi \vdash P$. By (ET-Par) and (ET-Nil) we have $\Delta, \Pi \vdash P \mid \mathbf{0}$.

2: By our inductive hypothesis we have that $\Delta, \Pi \vdash P \mid \mathbf{0}$. Then by (ET-Par) we have $\Delta, \Pi \vdash P$.

(**Par-A**): $\underbrace{P \mid (Q \mid R)}_{1} \equiv \underbrace{(P \mid Q) \mid R}_{2}$

1: By our inductive hypothesis we have that $\Delta, \Pi \vdash P \mid (Q \mid R)$. Then by two applications of (ET-Par) we have $\Delta, \Pi \vdash P$, $\Delta, \Pi \vdash Q$ and $\Delta, \Pi \vdash R$. Then we have $\Delta, \Pi \vdash (P \mid Q) \mid R$ by two applications of (ET-Par).

2: Similar to the first case.

(**Par-B**): $\underbrace{P \mid Q}_{1} \equiv \underbrace{Q \mid P}_{2}$

1: By our inductive hypothesis we have that $\Delta, \Pi \vdash P \mid Q$. By (ET-Par) we have $\Delta, \Pi \vdash P$ and $\Delta, \Pi \vdash Q$. Then we have $\Delta, \Pi \vdash Q \mid P$ by (ET-Par).

2: Similar to the first case.

(**Replicate**): $\underbrace{!P}_{1} \equiv \underbrace{P \mid !P}_{2}$

1: By our inductive hypothesis we have that $\Delta, \Pi \vdash !P$. By (ET-Rep) we that $\Delta, \Pi \vdash P$. Using (ET-Par) we have $\Delta, \Pi \vdash P \mid !P$.

2: By our inductive hypothesis we have that $\Delta, \Pi \vdash (P \mid !P$. By (ET-Rep) we that $\Delta, \Pi \vdash P$.

(**New-0**): $\underbrace{(\nu a : t).\mathbf{0}}_{1} \equiv \underbrace{\mathbf{0}}_{2}$

1: By our inductive hypothesis we have that $\Delta, \Pi \vdash (\nu a : t).\mathbf{0}$. By (ET-Res) we have that $\Delta, a : t, \Pi \vdash \mathbf{0}$ for some t. We can then prove $\Delta, \Pi \vdash 0$ using Lemma 3.7 as it states that we can remove names from the $\mathbf{0}$ process and still be well-typed.

2: By our inductive hypothesis we have that $\Delta, \Pi \vdash \mathbf{0}$. Using Lemma 3.5 we can show that $\Delta, a : t, \Pi \vdash \mathbf{0}$ and using (ET-Res) show that $\Delta, \Pi \vdash (\nu a : t).\mathbf{0}$ is still well-typed.

(**New-A**): $\underbrace{(\nu a : t_1).(\nu b : t_2).P}_{1} \equiv \underbrace{(\nu b : t_2).(\nu a : t_1).P}_{2}$

1: By our inductive hypothesis we have that $\Delta, \Pi \vdash \nu a.\nu b.P$. By applying (ET-Res) twice and we get $\Delta, a : \tau_1, b : \tau_2, \Pi \vdash P$ and $\Delta, b : \tau_2, a : \tau_1, \Pi \vdash P$ as the order is irrelevant $(\nu a : t_1).(\nu b : t_2).P$ is still well-typed.

2: Similar to the first case.

(**New-B**): $\underbrace{P \mid (\nu a : t).Q}_{1} \equiv \underbrace{(\nu a : t).(P \mid Q)}_{2}$

1: By our inductive hypothesis we have that $\Delta, \Pi \vdash P \mid \nu a.Q$ and by (ET-Par) we have $\Delta, \Pi \vdash P$ and $\Delta, \Pi \vdash (\nu a : t).Q$. The premise $\Delta, \Pi \vdash (\nu a : t).Q$ must be concluded with (ET-Res) giving us $\Delta, a : t, \Pi \vdash Q$. Then from the premise of (New-B) we have $a \notin \mathrm{fv}(P) \cup \mathrm{fn}(P)$. Then using Lemma 3.5 we get that $\Delta, a : t, \Pi \vdash P$ when $a \notin \mathrm{dom}(\Delta)$. Therefore, if $a \notin \mathrm{dom}(\Delta)$ it must hold that $\Delta, a : t, \Pi \vdash P$.

2: By our inductive hypothesis we have that $(\nu a : t).(P \mid Q)$. By Lemma 3.7 we have $\Delta, \Pi \vdash P$ from $\Delta, a : t, \Pi \vdash P$. We can then apply (ET-Res) on Q and (ET-Par).

□

D.6 Proof of Lemma 3.9

Proof of Lemma 3.10.

(**ET-N**): Then $T = n$. Then from the Definition 1.14 we get the substitution $n\left\{{}^{T'_u}/_{T_u}\right\} = n$ and applying (ET-N) preserves the typing $\Delta, \Pi \vdash n : \mathbf{Int}$.

(**ET-U**): Then $T = u$ and from (ET-U) we get that $\Delta, \Pi(u) = t$. Then from Definition 1.14 we get two cases:

$\boldsymbol{u = T_u}$: From the substitution we get that $u\left\{{}^{T'_u}/_{T_u}\right\} = T_{u'}$ and $u = T_u$ therefore $\Delta(T_u) = t$ and since $\Delta, \Pi \vdash T'_u : t_u$ we get that $t = t_u$. Then since $\Delta, \Pi \vdash T_u : t_u$

we get that $\Delta, \Pi(T_u) = t$. Therefore applying (ET-U) preserves the typing $\Delta, \Pi \vdash u\left\{{}^{T'_u}/_{T_u}\right\} : t$.

otherwise: From the substitution we get that $u\left\{{}^{T'_u}/_{T_u}\right\} = u$ and applying (ET-U) preserves the typing $\Delta, \Pi \vdash u : t$ and therefore $\Delta, \Pi \vdash u\left\{{}^{T'_u}/_{T_u}\right\} : t$ preservers the typing.

(**ET-Bin**): Then $T = T_1 \odot T_2$, where $\Delta, \Pi \vdash T_1 : \mathbf{Int}$,$\Delta, \Pi \vdash T_2 : \mathbf{Int}$ and $t = \mathbf{Int}$. Then from Definition 1.14 we get that $(T_1 \odot T_2)\left\{{}^{T'_u}/_{T_u}\right\} = T_1\left\{{}^{T'_u}/_{T_u}\right\} \odot T_2\left\{{}^{T'_u}/_{T_u}\right\}$. Then from the inductive hypothesis we get that $\Delta, \Pi \vdash T_1\left\{{}^{T'_u}/_{T_u}\right\} : \mathbf{Int}$ and $\Delta, \Pi \vdash T_2\left\{{}^{T'_u}/_{T_u}\right\} : \mathbf{Int}$. Therefore, applying (ET-Bin) preservers the typing $\Delta, \Pi \vdash (T_1 \odot T_2)\left\{{}^{T'_u}/_{T_u}\right\} : \mathbf{Int}$.

(**ET-Compx**): Then $T = c \cdot T'$ and as T' can be typed with either (ET-Compx) and (ET-Compn) depending on wether T' is x or n. Then from Definition 1.13 we get substitution on both c and T therefore we can apply it component wise s.t we get $c' = c\left\{{}^{T'_u}/_{T_u}\right\}$ and $T' = T\left\{{}^{T'_u}/_{T_u}\right\}$ from the induction hypothesis we get that $\Delta, \Pi(c') : @\ell$ and $\Delta(c', T') : \mathrm{pch}\left(\vec{t}\right)$. Since, the types $@\ell$ and $\mathrm{pch}\left(\vec{t}\right)$ are preserved after substitution then $\mathrm{pch}\left(\vec{t}\right) \in @\ell$ holds. Therefore, applying (ET-Compx) or (ET-Compn) preserves the typing $\Delta, \Pi \vdash (c \cdot T)\left\{{}^{T'_u}/_{T_u}\right\} : \mathrm{ch}(t)$.

□

D.7 Proof of Lemma 3.10

Proof of Lemma 3.9.

(**ET-Nil**): Then $P = \mathbf{0}$. Then from the Definition 1.15 we get the substitution $P\left\{{}^{T'_u}/_{T_u}\right\} = \mathbf{0}$ and using (ET-Nil) we get that $\Delta, \Pi \vdash P$.

(**ET-Par**): Then $P = Q \mid R$. Then using Definition 1.15 we get that $P\left\{{}^{T'_u}/_{T_u}\right\} = Q\left\{{}^{T'_u}/_{T_u}\right\} \mid R\left\{{}^{T'_u}/_{T_u}\right\}$. Then using the inductive hypothesis we get that $\Delta, \Pi \vdash Q\left\{{}^{T'_u}/_{T_u}\right\}$ and $\Delta, \Pi \vdash R\left\{{}^{T'_u}/_{T_u}\right\}$. Therefore, applying (ET-Par) preserves the typing $\Delta, \Pi \vdash P\left\{{}^{T'_u}/_{T_u}\right\}$.

(**ET-Rep**): Then $P = !Q$. Then from Definition 1.15 we get that $P\left\{{}^{T'_u}/_{T_u}\right\} = !\left(Q\left\{{}^{T'_u}/_{T_u}\right\}\right)$. Then using the inductive hypothesis we get that $\Delta, \Pi \vdash Q\left\{{}^{T'_u}/_{T_u}\right\}$. Therefore, applying (ET-Rep) preserves the typing $\Delta, \Pi \vdash P\left\{{}^{T'_u}/_{T_u}\right\}$.

(**ET-Res**): Then $P = (\nu a : t)Q$. Then from Definition 1.15 we get two cases:

$\boldsymbol{a \notin \mathbf{fn}(T_1) \cup \mathbf{fn}(T_2)}$: Then using the inductive hypothesis we get that $\Delta, a : t, \Pi \vdash\vdash Q\left\{{}^{T'_u}/_{T_u}\right\}$. Therefore, when applying (ET-Res) we get that $\Delta, \Pi \vdash (\nu a : t)Q\left\{{}^{T'_u}/_{T_u}\right\}$.

$\boldsymbol{a \notin \mathbf{fn}(T_1) \cup \mathbf{fn}(T_2)}$: From Definition 1.15 we get that $b \notin \text{fn}(T_u) \cup \text{fn}(T_{u'}) \cup \text{fn}(Q)$, $(\nu b : t)Q'\left\{{}^{T'''_u}/_{T''_u}\right\}$ and where $T''_u = T_u\{{}^{b}/_{a}\}$, $T'''_u = T'_u$ and $Q' = Q\{{}^{b}/_{a}\}$. Then by induction hypothesis we have $\Delta, b : t, \Pi \vdash Q\{{}^{b}/_{a}\}$ and the typing of Q' is preserved and by Lemma 3.10 the typing is preserved for T'' and T''' s.t $\Delta, b : t, \Pi \vdash T'' : t_u$ and $\Delta, b : t, \Pi \vdash T''' : t_u$. Then first case of Definition 1.15 applies as $b \notin \text{fn}(T''_u) \cup \text{fn}(T'''_u)$ therefore, using the induction hypothesis $\Delta, b : t, \Pi \vdash Q'$,$\Delta, b : t, \Pi \vdash T''_u : t_u$ and $\Delta, b : t, \Pi \vdash T'''_u : t_u$, . Then, when applying (ET-Res) preserves the typing $\Delta, \Pi \vdash (\nu b : t)Q'\left\{{}^{T'''_u}/_{T''_u}\right\}$.

(**ET-Send**): Then $P = \overline{c}\langle\vec{T}\rangle.Q$. Then from Definition 1.15 we get that $\left(\overline{c}\langle\vec{T}\rangle.Q\right)\left\{{}^{T'_u}/_{T_u}\right\} = \overline{c}\left\{{}^{T'_u}/_{T_u}\right\}\langle\vec{T}\left\{{}^{T'_u}/_{T_u}\right\}\rangle.Q\left\{{}^{T'_u}/_{T_u}\right\}$. Then using the inductive hypothesis we get that $\Delta, \Pi \vdash \overline{c}\left\{{}^{T'_u}/_{T_u}\right\}$, $\Delta, \Pi \vdash Q\left\{{}^{T'_u}/_{T_u}\right\}$, and $\Delta, \Pi \vdash \vec{T}\left\{{}^{T'_u}/_{T_u}\right\}$. Then, when applying (ET-Send) preserves the typing $\Delta, \Pi \vdash \overline{c}\langle\vec{T}\rangle.Q)\left\{{}^{T'_u}/_{T_u}\right\}$.

(**ET-Broad**): Then $P = \overline{c}{:}\langle\vec{T}\rangle.Q$. Then from Definition 1.15 we get that $\left(\overline{c}{:}\langle\vec{T}\rangle.Q\right)\left\{{}^{u_2}/_{u_1}\right\} = \overline{c}\left\{{}^{u_2}/_{u_1}\right\}\langle\vec{T}\left\{{}^{u_2}/_{u_1}\right\}\rangle.Q\left\{{}^{u_2}/_{u_1}\right\}$. Then using the inductive hypothesis we get that $\Delta, \Pi \vdash \overline{c}\left\{{}^{T'_u}/_{T_u}\right\}$, $\Delta, \Pi \vdash Q\left\{{}^{T'_u}/_{T_u}\right\}$, and $\Delta, \Pi \vdash \vec{T}\left\{{}^{T'_u}/_{T_u}\right\}$. Then, when applying (ET-Broad) preserves the typing $\Delta, \Pi \vdash \overline{c}{:}\langle\vec{T}\rangle.Q)\left\{{}^{u_2}/_{u_1}\right\}$.

(**ET-Recv**): Then $P = c\left(\vec{x} : \vec{t}\right).Q$. Then from Definition 1.15 we get two cases:

$\boldsymbol{\vec{x} \cap \mathbf{fv}(T_1) \cup \mathbf{fv}(T_2) = \emptyset}$: From Definition 1.15 we get that $\left(c\left(\vec{x} : \vec{t}\right).Q\right)\left\{{}^{T'_u}/_{T_u}\right\} = c\left\{{}^{T'_u}/_{T_u}\right\}\left(\vec{x} : \vec{t}\right).Q\left\{{}^{T'_u}/_{T_u}\right\}$. Then using the inductive hypothesis we get that $\Delta, \vec{x} : , \vec{t}, \Pi \vdash c\left\{{}^{T'_u}/_{T_u}\right\} : \text{ch}\left(\vec{t}\right)$,$\Delta, \vec{x} :, \vec{t}, \Pi \vdash Q\left\{{}^{T'_u}/_{T_u}\right\}$. Therefore, applying (ET-Recv) preserves the typing $\Delta, \Pi \vdash c\left(\vec{x} : \vec{t}\right).Q\left\{{}^{T'_u}/_{T_u}\right\}$

$\boldsymbol{\vec{x} \cap \mathbf{fv}(T_1) \cup \mathbf{fv}(T_2) \neq \emptyset}$: From Definition 1.15 we get that $\vec{y} \notin \text{fn}(T_u) \cup \text{fn}(T_{u'}) \cup \text{fn}(Q)$, $c\left(\vec{y} : \vec{t}\right).Q'$ and where $T''_u = T_u\left\{{}^{\vec{y}}/_{\vec{x}}\right\}$, $T'''_u = T'_u\left\{{}^{\vec{y}}/_{\vec{x}}\right\}$ and $Q' = Q\left\{{}^{\vec{y}}/_{\vec{x}}\right\}$. Then by induction hypothesis we have $\Delta, \vec{y} : \vec{t}, \Pi \vdash Q\left\{{}^{\vec{y}}/_{\vec{x}}\right\}$ and the typing of Q' is preserved and by Lemma 3.10 the typing is preserved for T'' and T''' s.t $\Delta, \vec{y} : \vec{t}, \Pi \vdash T'' : t_u$ and $\Delta, \vec{y} : \vec{t}, \Pi \vdash T''' : t_u$. Then first case of Definition 1.15 applies as $b \notin \text{fn}(T''_u) \cup \text{fn}(T'''_u)$ therefore, using the induction hypothesis $\Delta, \vec{y} : \vec{t} \vdash c\left\{{}^{T'''_u}/_{T''_u}\right\} : \text{ch}\left(\vec{t}\right)$, $\Delta, \vec{y} : \vec{t}, \Pi \vdash Q'\left\{{}^{T'''_u}/_{T''_u}\right\}$. Therefore, applying (ET-Recv) preserves the typing $\Delta, \Pi \vdash \left(c\left(\vec{y} : \vec{t}\right).Q\right)\left\{{}^{T'''_u}/_{T''_u}\right\}$

(**ET-Match**): Then $P = [T_1 \bowtie T_2]Q, R$ and from (ET-Match) we get that $\Delta, \Pi \vdash T_1 : t$, $\Delta, \Pi \vdash T_2 : t$, $\Delta, \Pi \vdash Q$ and $\Delta, \Pi \vdash R$. Then from Definition 1.15 we get that $P\left\{{}^{T'_u}/_{T_u}\right\} = \left[T_1\left\{{}^{T'_u}/_{T_u}\right\} \bowtie T_2\left\{{}^{T'_u}/_{T_u}\right\}\right]Q\left\{{}^{T'_u}/_{T_u}\right\}, R\left\{{}^{T'_u}/_{T_u}\right\}$. Then from Lemma 3.10 we get that

$\Delta, \Pi \vdash T_1\left\{{}^{T'_u}/_{T_u}\right\} : t$ and $\Delta, \Pi \vdash T_2\left\{{}^{T'_u}/_{T_u}\right\} : t$. Then by the induction hypothesis we get that $\Delta, \Pi \vdash Q\left\{{}^{T'_u}/_{T_u}\right\}$ and $\Delta, \Pi \vdash R\left\{{}^{T'_u}/_{T_u}\right\}$. Therefore, applying (ET-Match) preserves the typing $\Delta, \Pi \vdash ([T_1 \bowtie T_2]Q, R)\left\{{}^{T'_u}/_{T_u}\right\}$

□

D.8 Proof of Theorem 3.2 - subject reduction

Proof of Proof of subject reduction.

We will prove subject reduction by induction in the rule for concluding $P \to P'$.

(**E-Com**): By our assumption we know that $\Delta, \Pi \vdash \overline{c}\langle\vec{T}\rangle.P$ and $\Delta, \Pi \vdash c\left(\vec{x} : \vec{t}\right).Q$ by the application of (ET-Par). By (E-Com) we have $\overline{c}\langle\vec{T}\rangle.P | c\left(\vec{x}\right).Q \xrightarrow{\tau} P \mid Q\left\{{}^{\vec{T}}/_{\vec{x}}\right\}$ and must show that $\Delta, \Pi \vdash P \mid Q\left\{{}^{\vec{T}}/_{\vec{x}}\right\}$.

Then by (ET-Send) we have that $c : \mathrm{ch}\left(\vec{t}\right)$ and $\vec{T} : \vec{t}$, and by (ET-Recv) we have that $c : \mathrm{ch}\left(\vec{t}\right)$ and $\Delta, \vec{x} : \vec{t}, \Pi \vdash Q$. By using Lemma 3.9 we have that $\Delta, \vec{x} : \vec{t}, \Pi \vdash Q\left\{{}^{\vec{T}}/_{\vec{x}}\right\}$, and by Lemma 3.7 we have $\Delta, \Pi \vdash Q\left\{{}^{\vec{T}}/_{\vec{x}}\right\}$. We can therefore conclude $\Delta, \Pi \vdash P \mid Q\left\{{}^{\vec{T}}/_{\vec{x}}\right\}$ by (ET-Par).

(**E-Broad**): By our assumption we know that $\Delta, \Pi \vdash \overline{c}{:}\langle\vec{T}\rangle.Q$, $\Delta, \Pi \vdash c\left(\vec{x}_1 : \vec{t}\right).P_1$, ..., $\Delta, \Pi \vdash c\left(\vec{x}_n : \vec{t}\right).P_n$ given multiple applications of (ET-Par). By (E-Broad) we have $\overline{c}{:}\langle\vec{T}\rangle.Q \mid c\left(\vec{x}_1\right).P_1 | ... | c\left(\vec{x}_n\right).P_n \xrightarrow{:c} Q \mid P_1\left\{{}^{\vec{T}}/_{\vec{x}_1}\right\} \mid ... \mid P_n\left\{{}^{\vec{T}}/_{x_n}\right\}$ and must show that $\Delta, \Pi \vdash Q \mid P_1\left\{{}^{\vec{T}}/_{\vec{x}_1}\right\} \mid ... \mid P_n\left\{{}^{\vec{T}}/_{x_n}\right\}$.

Then by (ET-Broad) we have that $c : \mathrm{ch}\left(\vec{t}\right)$ and $\vec{T} : \vec{t}$, and by (ET-Recv) we have that $c : \mathrm{ch}\left(\vec{t}\right)$ and $\Delta, \vec{x}_i : \vec{t}, \Pi \vdash P_i$ for all $i \in \{1, ..., n\}$. By using Lemma 3.9 we have that $\Delta, \vec{x}_i : \vec{t}, \Pi \vdash P_i\left\{{}^{\vec{T}}/_{\vec{x}_i}\right\}$ for all $i \in \{1, ..., n\}$, and by Lemma 3.7 we have $\Delta, \Pi \vdash P_i\left\{{}^{\vec{T}}/_{\vec{x}_i}\right\}$ for all $i \in \{1, ..., n\}$. We can therefore conclude $\Delta, \Pi \vdash Q \mid P_1\left\{{}^{\vec{T}}/_{\vec{x}_1}\right\} \mid ... \mid P_n\left\{{}^{\vec{T}}/_{\vec{x}_n}\right\}$ by multiple applications of (ET-Par).

(**E-Par**): By our assumption we know that $\Delta, \Pi \vdash P \mid Q$ and given application of (ET-Par) we know $\Delta, \Pi \vdash P$ and $\Delta, \Pi \vdash Q$. Given (E-Par) we know that $P \mid Q \xrightarrow{\tau} P' \mid Q$ and must show that $\Delta, \Pi \vdash P' \mid Q$. By induction we get that $\Delta, \Pi \vdash P \xrightarrow{\tau} P'$ and thereby also for $\Delta, \Pi \vdash P'$. By (ET-Par) we then have $\Delta, \Pi \vdash P' \mid Q$.

(**E-Par2**): The proof is similar to (E-Par).

(**E-Res1**): By our assumption we know that $\Delta, \Pi \vdash (\nu a : t).P$ and by (ET-Res) we have $\Delta, a : t, \Pi \vdash P$, and given (E-Res1) we know that $(\nu a : t).P \xrightarrow{q} (\nu a : t).P'$. We must then show $\Delta, \Pi \vdash (\nu a : t).P'$. Then by induction and given (ET-Res) we have $\Delta, a : t, \Pi \vdash P'$.

(**E-Res2**): The proof is similar to (E-Res1).

(**E-Then**): By our assumption we know $\Delta, \Pi \vdash [T_1 \bowtie T_2]P, Q$ by (ET-Match) and $[T_1 \bowtie T_2]P, Q \xrightarrow{\tau} P$ by (E-Then). We must then show $\Delta, \Pi \vdash P$. This follows immediately by (ET-Match) as $\Delta, \Pi \vdash [T_1 \bowtie T_2]P, Q$ is only correct if $\Delta, \Pi \vdash P$.

(**E-Else**): By our assumption we know $\Delta, \Pi \vdash [T_1 \bowtie T_2]P, Q$ by (ET-Match) and $[T_1 \bowtie T_2]P, Q \xrightarrow{\tau} Q$ by (E-Else). We must then show $\Delta, \Pi \vdash Q$. This follows immediately by (ET-Match) as $\Delta, \Pi \vdash [T_1 \bowtie T_2]P, Q$ is only correct if $\Delta, \Pi \vdash Q$.

(**E-Struct**): By our assumption we know that $\Delta, \Pi \vdash P$ and $P \xrightarrow{q} P'$, and by Lemma 3.8 we have that $\Delta, \Pi \vdash Q$. By induction we have that $\Delta, \Pi \vdash Q'$ and by Lemma 3.8 we get $\Delta, \Pi \vdash P'$

□

D.9 Proof of Theorem 3.2 - type safety

Proof of Type safety of processes.

We prove type safety by induction in the type rules.

(**ET-Nil**): Trivial as for (ET-Nil) $P \not\mapsto \mathit{error}$

(**ET-Send**): By our assumption we have that $\Delta, \Pi \vdash \overline{c}\langle\vec{T}\rangle.P$ and must prove that $\overline{c}\langle\vec{T}\rangle.P \overset{\tau}{\not\mapsto} \mathit{error}$. By inspecting (ER-Send) we can see that $\overline{c}\langle\vec{T}\rangle.P \xrightarrow{\tau} \mathit{error}$ only if $c : \mathrm{ch}\left(\vec{t}_1\right)$, $\vec{T} : \vec{t}_2$ where $\vec{t}_1 \neq \vec{t}_2$. This contradicts the type rule for send, that states $c : \mathrm{ch}\left(\vec{t}\right)$ and $\vec{T} : \vec{t}$ meaning it must hold that $\vec{t}_1 = \vec{t}_2$ for send to be well-typed and therefore $P \not\mapsto \mathit{error}$.

(**ET-Broad**): By our assumption we have that $\Delta, \Pi \vdash \overline{c}{:}\langle\vec{T}\rangle.P$ and must prove that $\overline{c}{:}\langle\vec{T}\rangle.P \overset{:c}{\not\mapsto} \mathit{error}$. By inspecting (ER-Broad) we can see that $\overline{c}{:}\langle\vec{T}\rangle.P \xrightarrow{:c} \mathit{error}$ only if $c : \mathrm{ch}\left(\vec{t}_1\right)$, $\vec{T} : \vec{t}_2$ where $\vec{t}_1 \neq \vec{t}_2$. This contradicts the type rule for broadcast, that states $c : \mathrm{ch}\left(\vec{t}\right)$ and $\vec{T} : \vec{t}$ meaning it must hold that $\vec{t}_1 = \vec{t}_2$ for broadcast to be well-typed and therefore $P \not\mapsto \mathit{error}$.

(**ET-Recv**): By our assumption we have that $\Delta, \Pi \vdash c\left(\vec{x} : \vec{t}\right).P$ and must prove that $c\left(\vec{x} : \vec{t}\right).P \overset{\tau}{\not\mapsto} \mathit{error}$. By inspecting (ER-Recv) we can see that $c\left(\vec{x} : \vec{t}\right).P \xrightarrow{\tau} \mathit{error}$ if $c : \mathrm{ch}\left(\vec{t}_1\right)$ and $\vec{x} : \vec{t}_2$ where $\vec{t}_1 \neq \vec{t}_2$. This contradicts (ET-Recv) as for receive to be well-typed $c : \mathrm{ch}\left(\vec{t}\right)$ and $\vec{x} : \vec{t}$, and therefore it must be that $t_1 = t_2$. We can therefore conclude $P \not\mapsto \mathit{error}$.

(**ET-Match**): By our assumption we have that $\Delta, \Pi \vdash [T_1 \bowtie T_2]P, Q$ and must prove that $[T_1 \bowtie T_2]P, Q \overset{\tau}{\nrightarrow} \mathit{error}$. By inspecting (ER-Match) we can see that $[T_1 \bowtie T_2]P, Q \xrightarrow{\tau} \mathit{error}$ if $T_1 : t_1$, $T_2 : t_2$ and $t_1 \neq t_2$. This contradicts (ET-Match) as $T_1 : t$ and $T_2 : t$, that being $t_1 = t_2$ for match to be well-typed and therefore $P \nrightarrow \mathit{error}$

(**ET-Par**): By our assumption we have that $\Delta, \Pi \vdash P \mid Q$ and must prove that $P \mid Q \overset{\tau}{\nrightarrow} \mathit{error}$. We have two cases where we can go to the *error* process.

(**ER-Par1**): By inspecting (ER-Par1) we can see that $P \mid Q \xrightarrow{\tau} \mathit{error}$ if $P \xrightarrow{\tau} \mathit{error}$. Since $\Delta, \Pi \vdash P \mid Q$ we have that $\Delta, \Pi \vdash P$ by (ET-Par), and by our induction hypothesis we have $P \overset{\tau}{\nrightarrow} \mathit{error}$ and therefore we have a contradiction.

(**ER-Par2**): By inspecting (ER-Par2) we can see that $P \mid Q \xrightarrow{\tau} \mathit{error}$ if $Q \xrightarrow{\tau} \mathit{error}$. Since $\Delta, \Pi \vdash P \mid Q$ we have that $\Delta, \Pi \vdash Q$ by (ET-Par), and by our induction hypothesis we have $Q \overset{\tau}{\nrightarrow} \mathit{error}$ and therefore we have a contradiction.

(**ET-Res**): By our assumption we have that $\Delta, \Pi \vdash (\nu a : t).P$ and must prove that $(\nu a : t).P \overset{q}{\nrightarrow} \mathit{error}$. Since $\Delta, \Pi \vdash (\nu a : t).P$ we have that $\Delta, \Pi, a : t \vdash P$ by (ET-Par), and by our induction hypothesis we have $P \overset{\tau}{\nrightarrow} \mathit{error}$ and therefore we have a contradiction.

(**ET-Rep**): By our assumption we have that $\Delta, \Pi \vdash !P$ and must prove that $!P \overset{\tau}{\nrightarrow} \mathit{error}$. Since $\Delta, \Pi \vdash !P$ we have that $\Delta, \Pi \vdash P$ by (ET-Par), and by our induction hypothesis we have $P \overset{\tau}{\nrightarrow} \mathit{error}$ and therefore we have a contradiction.

□

Proof of Type safety of terms.

(**ET-N**): Trivial as for (ET-N) $P \nrightarrow \mathit{error}$

(**ET-U**): Trivial as for (ET-U) $P \nrightarrow \mathit{error}$

(**ET-Bin**): By our assumption we have that $\Delta, \Pi \vdash T_1 \odot T_2$ and must prove that $T_1 \odot T_2 \overset{\tau}{\nrightarrow} \mathit{error}$. We have two cases where we can go to the *error* process.

(**ER-Bin1**): By inspecting (ER-Bin1) we can see that $T_1 \odot T_2 \xrightarrow{\tau} \mathit{error}$ if $T_1 : t$ and $t \neq \mathbf{Int}$. This contradicts (ET-Bin) as for binary operation to be well-typed the type of T_1 must be **Int**. Therefore $P \nrightarrow \mathit{error}$

(**ER-Bin2**): By inspecting (ER-Bin2) we can see that $T_1 \odot T_2 \xrightarrow{\tau} \mathit{error}$ if $T_2 : t$ and $t \neq \mathbf{Int}$. This contradicts (ET-Bin) as for binary operation to be well-typed the type of T_2 must be **Int**. Therefore $P \nrightarrow \mathit{error}$

(**ET-Compx**): By our assumption we have that $\Delta, \Pi \vdash u \cdot x$ and must prove that $u \cdot x \overset{\tau}{\nrightarrow} \mathit{error}$. We have three cases where we can go to the *error* process.

(**ER-Compx1**): By inspecting (ER-Compx1) we can see that $u \cdot x \xrightarrow{\tau} \mathit{error}$ if u is not a location type. This contradicts (ET-Compx) as for a composite name to be well-typed u must be a location type. Therefore $P \nrightarrow \mathit{error}$.

(**ER-Compx2**): By inspecting (ER-Compx2) we can see that $u \cdot x \xrightarrow{\tau} \mathit{error}$ if $\Pi(u, x) \neq \mathrm{pch}\left(\vec{t}\right)$. This contradicts (ET-Compx) as for a composite name to be well-typed we have $\Delta(x) = \mathrm{pch}\left(\vec{t}\right)$. Therefore $P \nrightarrow \mathit{error}$.

(**ER-Compx3**): By inspecting (ER-Compx3) we can see that $u \cdot x \xrightarrow{\tau} \mathit{error}$ if $\mathrm{pch}\left(\vec{t}\right) \notin @\ell$. This contradicts (ET-Compx) as for a composite name to be well-typed we have $\mathrm{pch}\left(\vec{t}\right) \in @\ell$. Therefore $P \nrightarrow \mathit{error}$.

(**ET-Compn**): By our assumption we have that $\Delta, \Pi \vdash u \cdot x$ and must prove that $u \cdot x \overset{\tau}{\nrightarrow} \mathit{error}$. We have three cases where we can go to the *error* process.

(**ER-Compn1**): By inspecting (ER-Compn1) we can see that $u \cdot n \xrightarrow{\tau} \mathit{error}$ if u is not a location type. This contradicts (ET-Compn) as for a composite name to be well-typed u must be a location type. Therefore $P \nrightarrow \mathit{error}$.

(**ER-Compn2**): By inspecting (ER-Compn2) we can see that $u \cdot n \xrightarrow{\tau} \mathit{error}$ if $\Pi(u, n) \neq \mathrm{pch}\left(\vec{t}\right)$. This contradicts (ET-Compn) as for a composite name to be well-typed we have $\Pi(u, n) = \mathrm{pch}\left(\vec{t}\right)$. Therefore $P \nrightarrow \mathit{error}$.

(**ER-Compn3**): By inspecting (ER-Compn3) we can see that $u \cdot n \xrightarrow{\tau} \mathit{error}$ if $\mathrm{pch}\left(\vec{t}\right) \notin @\ell$. This contradicts (ET-Compn) as for a composite name to be well-typed we have $\mathrm{pch}\left(\vec{t}\right) \in @\ell$. Therefore $P \nrightarrow \mathit{error}$.

□

E Proofs for the Translation of BtF to TEπ

E.1 Proof of Lemma 4.1

Proof of Lemma 4.1.

We prove this using the cases where P appears in $C[P]$ unguarded or guarded.

***P* is guarded**: If P is guarded in $C[P]$ then only C can reduce and the first case applies.

***P* is unguared**: If P is unguarded in $C[P]$ then it has to be behind a combination of $!, \nu a, Q \mid$ in C. Using structural congruence we can construct the following two cases

$$C[P] \equiv \nu\vec{a}.(O \mid P) \text{ or } C[P] \equiv !\big(\nu\vec{a}.(O \mid P)\big)$$

Further using structural congruence we can show that we can transform the second case into the form of the first.

$$\begin{aligned} &!\big(\nu\vec{a}.(O \mid P)\big) && \equiv \\ &!\big(\nu\vec{a}.(O \mid P)\big) \mid \vec{b}.(O \mid P) && \equiv \\ &!\Big(\nu\vec{b}.\big(!\big(\nu\vec{a}.(O \mid P)\big) \mid O \mid P\big)\Big) && = \\ &!\nu\vec{b}.(O' \mid P) \text{ where } O' \equiv !\big(\nu\vec{a}.(O \mid P)\big) \end{aligned}$$

Therfore we get that Q is one of the following three cases.

$\boldsymbol{\nu\vec{a}.(O \mid P')}$: As $Q \equiv \nu\vec{a}.(O \mid P')$ we have that $P \xrightarrow{s} P'$ and since only P reduces we have that $Q \equiv C[P']$. Since, we have $C[P']$ and $P \xrightarrow{s} P'$ case (2) applies.

$\boldsymbol{\nu\vec{a}.(O' \mid P)}$: As $Q \equiv \nu\vec{a}.(O' \mid P)$ we have that $O \xrightarrow{s} O'$ and since only O reduces we have that $Q \equiv C'[P]$ where $C'[\cdot] = \nu\vec{a}.(O' \mid \cdot)$. Since no reduction on P occurs from the reduction $C[P] \xrightarrow{s} C'[P]$ we can conclude that the context can take the same reduction with the inactive process s.t $C[\mathbf{0}] \xrightarrow{s} C[\mathbf{0}]$. Then, as we have $C'[P]$, $O \xrightarrow{s} O'$ and $C[\mathbf{0}] \xrightarrow{s} C[\mathbf{0}]$ case (1) applies.

$\boldsymbol{\nu\vec{a}.(O' \mid P')}$: As $Q \equiv \nu\vec{a}.(O' \mid P')$ we have that both O and P reduces and that $Q \equiv C'[P']$. Then as both O and P are reduced in a single step it must be either a (E-Com) or (E-Broad). Therefore there must $\exists b$ s.t $O \downarrow_b$ and $P \downarrow_b$ then we have that $O \mid P \xrightarrow{s}= O' \mid P'$, and $C[P] \xrightarrow{s} C'[P']$. Then, we have $C[P']$,$O \mid P \xrightarrow{s}= O' \mid P'$ and $C[P] \xrightarrow{s} C'[P']$ and therefore case (3) applies.

□

E.2 Proof of Lemma 4.2

Before we start the proof of Lemma 4.2 we first need some important definitions. We have taken the same approach to prove Lemma 4.2 as Hüttel et. al and therefore the approach for the proof will be similar to the one seen in [2].

First we have the definition U which is the building blocks for the translation. By this we mean that for any $\Gamma \vdash e : \tau$ there should exist a process P and output channel o such that $[\![e]\!]_o^\Gamma \equiv P$ and that P is in the set of all possible U. As we know that e is well typed then by the proof of Theorem 4.1 we know that P is well-typed and therefore we will denote the types in some of the building blocks by t.

Definition 5.1 (Building blocks U): We define the set of translation building blocks U as follows.

$$\begin{aligned} U = {} & o(v:t).U \mid h(v:t_1, o:t_2).U \mid !h(v:t_1, o:t_2).U \mid h \cdot n(v:t).U \mid h \cdot \texttt{len}\ (n:t).U \mid \\ & h \cdot \texttt{tup}\ (v_1:t_1, ..., v_n:t_n).U \mid h \cdot \texttt{all}\ (c:t).U \mid !h \cdot \texttt{all}\ (c:t).U \mid \overline{h \cdot \texttt{all}}{:}\langle c \rangle.U \mid \\ & c(n:t_1, v:t_2).U \mid !c(n:t_1, v:t_2).U \mid d(\_:t).U \mid [n \neq 0]U, U \mid U|U \mid (\nu a:t).U \mid \\ & \overline{o}\langle v \rangle \mid \overline{h}\langle v, o \rangle \mid \overline{h \cdot n}\langle n, v \rangle \mid !\overline{h \cdot n}\langle n, v \rangle \mid \overline{h \cdot \texttt{len}}\langle n \rangle \mid \overline{h \cdot \texttt{tup}}\langle v_1, ..., v_n \rangle \mid \\ & !\overline{h \cdot \texttt{tup}}\langle v_1, ..., v_n \rangle \mid \overline{c}\langle n, v \rangle \mid \overline{d}\langle 0:t \rangle \mid \mathit{Repeat}(n, c, d) \mid \mathbf{0} \end{aligned}$$

we denote the set of all possible U as $\mathcal{U}$.

In the translation we have 4 categories of channels $o \in \Omega$, $h \in \Lambda$, $d \in \mathrm{K}$, $c \in \Psi$ and $\mathcal{A} = \Omega \cup \Lambda \cup \mathrm{K} \cup \Psi$ where $a \in \mathcal{A}$. We denote $\mathcal{P} = \{P_1, ..., P_n\}$ as a set of processes in U where $\forall i \in \{1..n\}.P_i \downarrow_h$, and therefore $\mathcal{P} \subseteq \mathcal{U}$. We denote $\mathcal{Q} = \{Q_1, ..., Q_n\}$ as the set of processes s.t for some $Q_i \in \mathcal{Q}$, Q_i communicates over h_i instead of h.

Definition 5.2 (Client function f): We create a function that partially maps from sub processes to the powerset of processes $f : \mathcal{Q} \rightharpoonup \mathbb{P}(\mathcal{P})$

This function takes a sub-process Q and then returns the processes Q communicates with, that being Q's clients. Next we define a new relation R that relates processes with a single handle for communication with a Q to processes with multiple handles for communication with multiple Q's. In this relation C is a complete context (see Definition 4.2).

Definition 5.3 (Relation R):

$$R = \Bigg\{ \Bigg(C\Bigg[(\nu h : t_h).\Big(\nu \mathcal{A} : \vec{t}_a\Big).\Bigg(Q \mid \prod_{P_i \in \mathcal{P}} P_i \mid U\Bigg)\Bigg], \\ C\Bigg[\Big(\nu h_1 : t_{h_1}\Big). \ldots .\Big(\nu h_n : t_{h_n}\Big).\Big(\nu \mathcal{A} : \vec{t}_a\Big).\Bigg(\prod_{Q_i \in \mathcal{Q}} \mid \prod_{Q_j \in \mathcal{Q}} \prod_{P_i \in f(Q_j)} P_i\Big\{{}^{h}/_{h_j}\Big\} \mid U\Bigg)\Bigg]\Bigg) \mid \\ U \not\downarrow_h \text{ and } \forall i \in \{1...n\}.U \not\downarrow_{h_i} \text{ and } \Bigg(\bigcup_{Q_i \in \mathcal{Q}} f(Q_i)\Bigg) = \mathcal{P} \text{ and } \\ \forall Q_i, Q_j. \text{ where } i \neq j \text{ then } f(Q_i) \cap f\big(Q_j\big) = \emptyset \text{ and } \forall a \in \mathcal{A}.(\nu a : t_a).U \not\downarrow_a \Bigg\}$$

This brings us to the proof of Lemma 4.2 which states that for two well-typed BtF expressions and their

Proof of Lemma 4.2.

In order to prove Lemma 4.2 we have to prove that independent of what value e_1 is that $[\![e_1]\!]_o^\Gamma\{{}^{x}/_{h}\} \dot{\approx}_\alpha [\![e_1\{x \mapsto e_2\}]\!]_o^\Gamma$ by showing that R is a WABB and therefore closed under Definition 4.3. We do this by matching the transitions for each translated expression and showing that the new pair is in R.

e_2 is an integer: When e_2 is a number we have from the translation that $[\![n]\!]_o^\Gamma = \overline{o}\langle n\rangle$ where $n : \mathbf{Int}$, $o : \mathrm{ch}(\mathbf{Int})$ and $x : \mathbf{Int}$. As x is a variable the substitution is on translations of x in e_1. Therefore we show that $[\![x]\!]_o^\Gamma\{{}^{n}/_{x}\} \dot{\approx}_\alpha [\![x\{x \mapsto n\}]\!]_o^\Gamma$. Then we have the translation for x is $[\![x]\!]_o^\Gamma = \overline{o}\langle x\rangle$ and $[\![x\{x \mapsto n\}]\!]_o^\Gamma = \overline{o}\langle n\rangle$. We then have that $\overline{o}\langle x\rangle\{{}^{n}/_{x}\} = \overline{o}\langle n\rangle$ and since $\overline{o}\langle n\rangle \dot{\approx}_\alpha \overline{o}\langle n\rangle$ then $\overline{o}\langle x\rangle\{{}^{n}/_{x}\} \dot{\approx}_\alpha [\![x\{x \mapsto n\}]\!]_o^\Gamma$.

e_2 is an abstraction, **tuple or array**: For the process $[\![e_1\{x \mapsto e_2\}]\!]_o^\Gamma$ there can be different variations of processes from the translation of e_2. In this case a handle is used for communication with the translation of e_2.

1. $\boldsymbol{C[S] \to C'[S]}$: In this case we have an internal reduction in the context of the form $C[S] \to C'[S]$ where $S = (\nu h : t_h).(\nu \mathcal{A} : t_a).\Bigg(Q \mid \prod_{P_i \in \mathcal{P}} P_i \mid U\Bigg)$ s.t C can take the transition without S and behaves like $C[\mathbf{0}] \to C'[\mathbf{0}]$. Therefore we can use C' on the right side of the pair since it is not dependent on the process in the context. Furthermore since the restrictions on the relation only are on the inner process of the context the new pair is in R.

2. $\boldsymbol{U \to U'}$: In this case we have an internal transition inside U s.t $U \to U'$ and since U is the same on the left and right side of the pair there is a matching transition per Definition 4.3. However to show that U' is allowed in R we have to prove the following holds $U' \not\downarrow_h$, $\forall i \in \{1...n\}.U' \not\downarrow_{h_i}$ and $\forall a \in \mathcal{A}.(\nu a : t_a).U' \not\downarrow_a$. To prove the first

condition $U' \not\downarrow_h$ we destruct U' into it's components $U' = U_1 \mid ... \mid U_n$. Then to prove $U' \not\downarrow_h$ we have to construct a U'' as from the building blocks of U we have that there can exists a U_i of the form $U_i \notin \{!U, (\nu a : t_a).U, [n \neq 0]U, U, \mathbf{0}\}$ s.t that $U_i \downarrow_h$ and therefore $U' \downarrow_h$ which is not allowed in R. However we can construct a new U'' and $\mathcal{P}'$ s.t $U'' \not\downarrow_h$ and $\left(\bigcup_{Q_i \in \mathcal{Q}} f(Q_i)\right) = \mathcal{P}'$. We construct $\mathcal{P}'$ as $\mathcal{P}$ and the set of all sub-processes of U' where $U_i \downarrow_h$ then we construct U'' as the all the sub-processes of U' where $U_i \not\downarrow_h$ in parallel composition. Then as $U'' \not\downarrow_h$ and all of new exposed clients of $\mathcal{Q}$ has been moved to $\mathcal{P}'$ we have $\left(\bigcup_{Q_i \in \mathcal{Q}} f(Q_i)\right) = \mathcal{P}'$.

Then for the second condition $\forall i \in \{1...n\}.U'' \not\downarrow_{h_i}$ we use that on both sides of the pair U is the same. Then as we are only able to observer communication h_i after substituting h we know from the construction of U'' above that $U'' \not\downarrow_h$. Therefore we can only observe communication on h_i outside of U'' and therefore the condition $\forall i \in \{1...n\}.U'' \not\downarrow_{h_i}$ holds.

Then for the case of $\forall a \in \mathcal{A}.(\nu a : t_a).U'' \not\downarrow_a$ we can use structural congruence to move out all restrictions and therefore we have that the new pair is in R.

$$\left(C\left[(\nu h : t_h).\left(\nu \mathcal{A} : \vec{t}_a\right).\left(Q \mid \prod_{P_i' \in \mathcal{P}'} P_i' \mid U''\right)\right],\right.$$
$$\left. C\left[\left(\nu h_1 : t_{h_1}\right). \ ... \ .\left(\nu h_n : t_{h_n}\right).\left(\nu \mathcal{A} : \vec{t}_a\right).\left(\prod_{Q_i \in \mathcal{Q}} \mid \prod_{Q_j \in \mathcal{Q}} \prod_{P_i' \in f(Q_j)} P_i'\left\{{}^{h}/_{h_j}\right\} \mid U''\right)\right]\right)$$

3. $\boldsymbol{Q \mid P_i \to Q' \mid P_i'}$: For this case we consider the different forms of e_2 and Q where e_2 is either an abstraction, tuple or array.

$\boldsymbol{e_2 = \lambda(x : t).e_\lambda}$: Then $Q = !h(x, r).[\![e_\lambda]\!]_r^\Gamma$ and as P_i communicates with Q on h it has the following form $P_i = !\overline{h}\langle x, r\rangle.S$ where S is some continuation of P_i. Then we have the transition to $Q \mid P \to [\![e_\lambda]\!]_r^\Gamma \mid S$ in the right pair we can match this by using $P_i\left\{{}^{h}/_{h_j}\right\}$ where $P_i \in f(Q_j)$ s.t $Q_j \downarrow_{h_j}$. After the communication S is uncovered and a $[\![e_\lambda]\!]_r^\Gamma$ is spawned and added to $U' = U \cup [\![e_\lambda]\!]_r^\Gamma \cup S$. However as it is possible that $[\![e_\lambda]\!]_r^\Gamma \downarrow_h$ and $S \downarrow_h$ we use structural congruence as in the second case to find a U'' and P' s.t $U'' \not\downarrow_h$ and then the following holds:

$$Q \mid \prod_{P_i \in \mathcal{P}} P_i \mid U \to Q \mid \prod_{P_i \in \mathcal{P}'} P_i \mid U''$$

Therefore the pair after the transition is still in R.

$\boldsymbol{e_2 = (e_{2,1}, ..., e_{2,n})}$: Then $Q = !\overline{h \cdot \texttt{tup}}\langle T_1, ..., T_n\rangle \mid \overline{o}\langle h\rangle$ and as P_i communicates with Q on h it has the following form $P_i = h \cdot \texttt{tup}\ (v_1 : t_1, ..., v_n : t_n).S$ where S is

some continuation of P_i. Then using the same argument for the case of abstraction with $U' = U \cup S$.

$\boldsymbol{e_2} = [\boldsymbol{e_{2,1}}, ..., \boldsymbol{e_{2,n}}]$: Then $Q = \prod_{i \in 1..n} \left(!h \cdot \texttt{all}\ (r).\overline{r}\langle T_i \rangle \mid \overline{h \cdot i}\langle T_i \rangle\right) \mid \overline{o}\langle h \rangle$ and as P_i communicates with Q on h it has one of the following forms where S is some continuation of P_i.

$$P_i \in \left\{h \cdot i(v : t).S, \overline{h \cdot \texttt{all}}\langle h_x \rangle.!h_x(i : \textbf{Int}, v : t).S, h \cdot \texttt{len}\ (v : \textbf{Int}).S\right\}$$

Then using the same argument for the case of abstraction with $U' = U \cup S$.

Next we look at the right side of the pair in the relation.

1. In this case we have an internal communication in the context C. The proof follows the same argument as in case 1 above.
2. In this case we have an internal communication in U. The proof follows the same argument as in case 2 above.
3. For this case we consider the different forms of Q, that being e_1 is either an abstraction, tuple or array. The proof follows the same argument as in case 3 above communicating on some h_j instead.

The pair $(\llbracket \boldsymbol{e_1} \rrbracket^{\boldsymbol{\Gamma}}_{\boldsymbol{o}}\{^{\boldsymbol{x}}/_{\boldsymbol{h}}\}, \llbracket \boldsymbol{e_1}\{\boldsymbol{x} \mapsto \boldsymbol{e_2}\} \rrbracket^{\boldsymbol{\Gamma}}_{\boldsymbol{o}})$:

Lastly we show that R is closed under Definition 4.3 when on the form of the pair.

$$((\nu h : t).(Q \mid \llbracket e_1 \rrbracket^{\Gamma}_{o}\{^{x}/_{h}\}), \llbracket e_1\{x \mapsto e_2\} \rrbracket^{\Gamma}_{o})$$

In e_1 there exist n usages of x where each usage in $\llbracket e_1 \rrbracket^{\Gamma}_{o}$ is replaced with $\overline{o_1}\langle x \rangle, ..., \overline{o_n}\langle x \rangle$ where $\forall i \in \{1, ..., n\}.\overline{o_i}\langle x \rangle \in \mathcal{U}$. Then in $e_1\{x \mapsto e_2\}$ each use of x is replaced with the full expression e_2 s.t in $\llbracket e_1 \rrbracket^{\Gamma}_{o}$ each x is on the form $\forall i \in \{1, ..., n\}.\llbracket e_i \rrbracket^{\Gamma}_{o_i} = (\nu h : \tau).(Q \mid \overline{o_i}\langle h_i \rangle)$ where $\llbracket e_i \rrbracket^{\Gamma}_{o_i} \in \mathcal{U}$. Therefore $\llbracket e_1\{x \vdash e_2\} \rrbracket^{\Gamma}_{o} = \llbracket e_2 \rrbracket^{\Gamma}_{o_1} \mid ... \mid \llbracket e_2 \rrbracket^{\Gamma}_{o_n} \mid S$ and $\llbracket e_1 \rrbracket^{\Gamma}_{o}\{^{x}/_{h}\} = \overline{o_1}\langle h \rangle | \ ... \ | \overline{o_n}\langle h \rangle \mid S$ where S is the rest of $\llbracket e_2 \rrbracket^{\Gamma}_{o}$. We start by showing we can match transitions.

WABB: We start by expanding the translation of $\llbracket e_1\{x \mapsto e_2\} \rrbracket^{\Gamma}_{o} = Q_1 \mid \overline{o_1}\langle h_1 \rangle \mid ... \mid Q_n \mid \overline{o_n}\langle h_n \rangle \mid S$ and $\llbracket e_1 \rrbracket^{\Gamma}_{o}\{^{h}/_{x}\} = Q \mid \overline{o_1}\langle h \rangle \mid ... \mid \overline{o_n}\langle h \rangle \mid S$. The by inspection of the translation we have that $\forall i \in \{1, ..., n\}.(Q \equiv Q_i)$ as Q and Q_i are both translations of e_2 with different handles and therefore we can apply the structural congruence rule for alpha conversion. Furthermore from the translation we have that for each $Q \mid ... \mid \overline{o_i}\langle h \rangle$ on the left side of the pair there is a corresponding $Q_i \mid \overline{o_i}\langle h_i \rangle$ on the right side and as $e_2 \in \mathcal{V}$ we no there are no further important transitions. Then as S is the same on both sides of the pair we can match any internal transition of S.

Then we show that for the left and right side of the pair that the conditions for R is fulfilled.

Left side: Then for the left side we show that we can transform the form of the translation s.t it fulfills that $U \not\downarrow_h$ and $\forall a \in \mathcal{A}.(\nu a : t_a).U \not\downarrow_a$. To do this we

first find a U, Q and $\mathcal{P}$ s.t we can transform the process $W_1 = (\nu h : t).(\nu \mathcal{A} : t_a).(Q' \mid \overline{o_1}\langle h\rangle| \ ... \ |\overline{o_n}\langle h\rangle \mid S)$ to the following process.

$$(\nu h : t).(\nu \mathcal{A} : t_a).\left(Q \mid \prod_{P_i \in \mathcal{P}} P_i \mid U\right)$$

The first step is to abuse that all bindings of names are unique and therefore we can move out all bindings using structural congruence s.t we have the process on the following form.

$$W_1 \equiv (\nu h : t).\left(\nu \mathcal{A} : \vec{t_a}\right).(Q' \mid \overline{o_1}\langle h\rangle| \ ... \ |\overline{o_n}\langle h\rangle \mid S)$$

Then for every sub-process in S we check if $S_i \downarrow_h$ and when it is the case add it to $\mathcal{P}$ and otherwise to U and then we add $Q'|\overline{o_1}\langle h\rangle| \ ... \ |\overline{o_n}\langle h\rangle$ to Q. Therefore we have that $U \not\downarrow_h$ and $\forall a \in \mathcal{A}.(\nu a : t_a).U \not\downarrow_a$, and by the construction of U we have not added any process that communicates on h and as all restrictions have been moved to the top of the process. Therefore we can reconstruct W_1 using U and $\mathcal{P}$ s.t we have the following process.

$$(\nu h : t).(\nu \mathcal{A} : t_a).\left(Q \mid \prod_{P_i \in \mathcal{P}} P_i \mid U\right)$$

Right side: Then for the right side we show that we can transform the form of the translation s.t it fulfills that $\forall i \in \{1...n\}.U' \not\downarrow_{h_i}$, $\left(\bigcup_{Q_i \in \mathcal{Q}} f(Q_i)\right) = \mathcal{P}$, $\forall Q_i, Q_j.$ where $i \neq j$ then $f(Q_i) \cap f\big(Q_j\big) = \emptyset$ and $\forall a \in \mathcal{A}.(\nu a : t_a).U \not\downarrow_a$ To do this we first find a U, $\mathcal{Q}$ and $\mathcal{P}$ s.t we can transform the process $W_2 = (\nu h_1 : t).(Q_1 \mid \overline{o_1}\langle h_1\rangle) \mid ... \mid (\nu h_n : t).(Q_n \mid \overline{o_n}\langle h_n\rangle) \mid S$ to the following process.

$$\left(\nu h_1 : t_{h_1}\right). \ ... \ .\left(\nu h_n : t_{h_n}\right).(\nu \mathcal{A} : t_a).\left(\prod_{Q_i \in \mathcal{Q}} \mid \prod_{Q_j \in \mathcal{Q}} \prod_{P_i \in f(Q_j)} P_i\left\{{}^{h}/_{h_j}\right\} \mid U\right)$$

The first step is to abuse that all bindings of names are unique and therefore we can move them to start of the process s.t we have the process on the following form.

$$W_1 \equiv (\nu h_1 : t).....(\nu h_n : t).(Q_1 \mid \overline{o_1}\langle h_1\rangle) \mid ... \mid (Q_n \mid \overline{o_n}\langle h_n\rangle) \mid S$$

Then for every sub-process in S we check if $S_i \downarrow_h$ and if it is the case we add it to $\mathcal{P}$ and otherwise U. Then we construct $\mathcal{Q} = \{Q_1| \ \overline{o_1}\langle h_1\rangle, ..., Q_n| \ \overline{o_n}\langle h_n\rangle\}$. Therefore we have that $\forall i \in \{1...n\}.U \not\downarrow_{h_i}$ and $\left(\bigcup_{Q_i \in \mathcal{Q}} f(Q_i)\right) = \mathcal{P}$ holds as all clients of $\mathcal{Q}$ has been moved to $\mathcal{P}$ and $\forall Q_i, Q_j.$ where $i \neq j$ then $f(Q_i) \cap f\big(Q_j\big) = \emptyset$ holds as each $P_i \in \mathcal{P}$ receives their handle through some o_i specific to each instance of the translation of e_2. Lastly $\forall a \in \mathcal{A}.(\nu a : t_a).U \not\downarrow_a$ as all restrictions have been moved to the top of the process.

Therefore as the pair (W_1, W_2) fulfills the conditions for R and R is closed under Definition 4.3 with the pair it must hold for this case.

□

E.3 Proof of Lemma 4.3

Proof of Lemma 4.3.

We use $\xrightarrow{s}{}^{n}$ to denote n reductions of the form $\xrightarrow{s}$ then using induction on n.

$\boldsymbol{n = 0}$: In this case no reduction has occured and therefore $P' = P$ and we must show that $P\not\downarrow_\alpha$ for any prefix. We show this by contradiction and therefore assume there exists an α s.t $P\downarrow_\alpha$. From the assumption of $P \dot{\approx}_\alpha \mathbf{0}$ we have that $(P, \mathbf{0}) \in R$ and by the condition (4) of Definition 4.3 then it must hold that $\mathbf{0} \overset{\circ}{\Longrightarrow}\downarrow_\alpha$. However, since there are no reductions from $\mathbf{0}$ therefore $\mathbf{0} \overset{s}{\not\longrightarrow}$ and $\mathbf{0}\not\downarrow_\alpha$.

$\boldsymbol{n > 0}$: From the induction hypothesis we have $P \xrightarrow{s}{}^{n-1} P^{n-1}$ then $\forall\alpha.P^{n-1}\not\downarrow_\alpha$ and $P^{n-1} \dot{\approx}_\alpha \mathbf{0}$. Then we we will show that $P^{n-1} \xrightarrow{s} P'$ then $\forall\alpha.P'\not\downarrow_\alpha$ and $P' \dot{\approx}_\alpha \mathbf{0}$. Since $P^{n-1} \dot{\approx}_\alpha \mathbf{0}$ we know that there exist and R s.t $(P^{n-1}, \mathbf{0}) \in R$ and R is a Definition 4.3. Since $\mathbf{0} \overset{\bullet}{\not\longrightarrow}$, then $P \overset{\bullet}{\not\longrightarrow}$ therefore $P^{n-1} \xrightarrow{\circ} P'$. Then because $(P^{n-1}, \mathbf{0}) \in R$, then $(P', \mathbf{0}) \in R$, and therefore $P' \dot{\approx}_\alpha \mathbf{0}$ and $\forall\alpha.P\not\downarrow_\alpha$

□

E.4 Proof of Lemma 4.4

Proof of Lemma 4.4.

Let R be a relation $R = \{(C[P \mid Q], C[Q]) \mid P, Q \in \mathcal{P}, C \in \mathcal{C}, P \dot{\approx}_\alpha \mathbf{0}\}$. Then by Lemma 4.3 if $P \dot{\approx}_\alpha \mathbf{0}$ then $P\not\downarrow_\alpha$ therefore since $P \dot{\approx}_\alpha \mathbf{0}$ it holds that when $P \mid Q \downarrow_\alpha$ then $P\not\downarrow_\alpha$ and $Q \downarrow_\alpha$. We then prove that R is a Definition 4.3.

(**1**) **and** (**2**): By Definition 4.3 condition (1) and (2) if $C[P|Q] \xrightarrow{s} O$ then $C[Q] \overset{s}{\Longrightarrow} O'$ s.t $(O, O') \in R$. By Lemma 4.1 O is one of the following three cases.

***C* reduces alone**: Then $O = C'[P \mid Q]$. Here $C[Q]$ can follow by $C[Q] \xrightarrow{s} C'[Q]$.

***P* | *Q* reduces alone**: Then $O = C[R]$, where $R = P \mid Q \xrightarrow{s} R$. Since by Lemma 4.3 we have $P\not\downarrow_\alpha$, then either P or Q reduces alone.

$\boldsymbol{R = P' \mid Q}$: Then $P \xrightarrow{s} P'$, and because $P \dot{\approx}_\alpha \mathbf{0}$, $s = \circ$. Therefore $C[Q]$ can follow with no reductions as by Lemma 4.3 $P' \dot{\approx}_\alpha \mathbf{0}$ and $(C[P' \mid Q], C[Q]) \in R$

$\boldsymbol{R = P \mid Q'}$: Then $Q \xrightarrow{s} Q'$, $C[Q] \xrightarrow{s} C[Q']$ and $(C[P \mid Q'], C[Q']) \in R$

***P* | *Q* and *C* reduces**: Then $\exists R$ s.t $R \in C$ and $R \mid P \mid Q \xrightarrow{s} S' \mid P \mid Q'$ as by Lemma 4.3 we have $P\not\downarrow_\alpha$ and therefore P can not communicate with S. Therefore we have that $S \mid Q \xrightarrow{s} S' \mid Q'$ and then $C[Q] \xrightarrow{s} C'[Q']$.

(**3**): By the selection of R it holds.

(**4**): If $C[P \mid Q] \downarrow_\alpha$ then $Q \overset{\circ}{\Longrightarrow}\downarrow_\alpha$. By Lemma 4.3 then $P \not\downarrow_\alpha$, $Q \downarrow_\alpha$ and by Lemma 4.1 O is one of the following cases.

***C* reduces alone**: Then $O = C'[P \mid Q]$. Here $C[Q]$ can follow by $C[Q] \xrightarrow{s} C'[Q]$.

***Q* reduces alone**: Then $O = C[Q']$ where $Q \xrightarrow{s} Q'$ and $C[P \mid Q] \xrightarrow{s} C[P \mid Q']$.

***Q* and *C* reduces**: Follows the same argumentation as (1) and (2).

□

E.5 Proof of Lemma 4.5

Before we can prove Lemma 4.5 we need to know the depth of an expression e.

Definition 5.4 (Depth of expression): Let $\mathcal{D}(e)$ denote the depth of e.

$$\begin{aligned}
\mathcal{D}(n) &= 0 \\
\mathcal{D}(\lambda x.e) &= 0 \\
\mathcal{D}((e_1, ..., e_n)) &= \max_{i \in \{0,...,n\}} (\mathcal{D}(e_i)) + 1 \\
\mathcal{D}([e_1, ..., e_n]) &= \max_{i \in \{1,...,n\}} (\mathcal{D}(e_i)) + 1 \\
\mathcal{D}(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) &= \max_{i \in \{1,...,3\}} (\mathcal{D}(e_i)) + 1 \\
\mathcal{D}(e_1 \odot e_2) &= \max_{i \in \{1,...,2\}} (\mathcal{D}(e_i)) + 1 \\
\mathcal{D}(\texttt{size } e_1) &= \mathcal{D}(e_1) + 1 \\
\mathcal{D}(\texttt{iota } e_1) &= \mathcal{D}(e_1) + 1 \\
\mathcal{D}(\texttt{map } e_1) &= \mathcal{D}(e_1) + 1
\end{aligned}$$

Proof of Lemma 4.5.

We prove this on induction on the depth $\mathcal{D}(e)$ where $e \in \mathcal{V}$. As per (Adm) any $\xrightarrow{\tau}$ is an administrative reduction.

$\boldsymbol{\mathcal{D}(e) = 0}$: From Definition 5.4, we get that e must be either a number or abstraction.

n: From Chapter 4.1.1.1 $[\![n]\!]_o^\Gamma = \overline{o}\langle n \rangle$ can immediately send n on o and therefore $[\![n]\!]_o \downarrow_{\overline{o}}$

x: From Chapter 4.1.1.1 $[\![x]\!]_o^\Gamma = \overline{o}\langle x \rangle$ can immediately send x on o and therefore $[\![x]\!]_o \downarrow_{\overline{o}}$

λx.e: From Chapter 4.1.1.1 $[\![\lambda(x : \tau_1).e_1]\!]_o^\Gamma = (\nu h : [\![\tau_1 \to \tau_2]\!]).(o\langle h \rangle \mid !h(x : [\![\tau_1]\!], r : \text{ch}([\![\tau_2]\!])).[\![e_1]\!]_r^\Gamma)$ can send the handle h on o by going under restriction and the parallel composition therefore $(\nu h : [\![\tau_1 \to \tau_2]\!]).(o\langle h \rangle \mid !h(x : [\![\tau_1]\!], r : \text{ch}([\![\tau_2]\!])).[\![e_1]\!]_r^\Gamma) \downarrow_{\overline{o}}$

$\boldsymbol{\mathcal{D}(e) > 0}$: Since $\mathcal{D}(e) > 0$ then e must be either a tuple or array. From Definition 5.4 we get that the depth for tuple and array is $\mathcal{D}(e) = \max_{i \in \{0,...,n\}}(\mathcal{D}(e_i)) + 1$. Since e is a

value on the form $(e_1, ...e_n)$ or $[e_1, ..., e_n]$, then each e_i we have that for all $e_1, ..., e_n$ $\forall i \in \{1, ..., n\}$ where $\mathcal{D}(e_i) < \mathcal{D}(e)$. Then by the induction hypothesis we have that $[\![e_i]\!]o_i \overset{\circ}{\Longrightarrow} P_i$ where $P_i \downarrow_{\overline{o}_i}$

$(\boldsymbol{e_1}, ...\boldsymbol{e_n})$: From Chapter 4.1.1.1 we get the translation for array $[\![[e_1, ..., e_n]]\!]_o^\Gamma$ is

$$(\nu o_1 : \mathrm{ch}([\![\tau]\!])).....(\nu o_n : \mathrm{ch}([\![\tau]\!])).(\nu h : [\![[\tau]]\!]).$$
$$\left(\prod_{i=1}^{n} [\![e_i]\!]_{o_i}^\Gamma \mid o_1(v_1 : [\![\tau]\!]).....o_n(v_n : [\![\tau]\!]).\right.$$
$$\left.\left(\prod_{i=1}^{n} Cell(h, i-1, v_i, [\![\tau]\!]) \mid \overline{h \cdot \texttt{len}}\langle n\rangle \mid \overline{o}\langle h\rangle\right)\right)$$

Then by repeated application of (E-Res1) followed by (E-Com) therefore $q = \tau$ as communication reduction is a τ reduction. This then results in the following process $P = \left(\prod_{i=1}^{n} Cell(h, i-1, v_i) \mid \overline{h \cdot \texttt{len}}\langle n\rangle \mid \overline{o}\langle h\rangle\right)$ and as it was reached using only administrative reductions we have that $[\![[e_1, ..., e_n]]\!]_o^\Gamma \overset{\circ}{\Longrightarrow} P$. Then because P can send the handle h on o by going under the parallel composition we have that $P \downarrow_{\overline{o}}$

$[\boldsymbol{e_1}, ...\boldsymbol{e_n}]$: From Chapter 4.1.1.1 we get the translation for tuple $[\![(e_1, ..., e_n)]\!]_o^\Gamma$ is

$$(\nu o_1 : \mathrm{ch}([\![\tau_1]\!])).....(\nu o_n : \mathrm{ch}([\![\tau_n]\!])).$$
$$\left(\prod_{i=1}^{n} [\![e_i]\!]_{o_i}^\Gamma \mid o_1(v_1 : [\![\tau_1]\!]). \ ... \ .o_n(v_n : [\![\tau_n]\!]).\right.$$
$$\left.(\nu h : [\![(\tau_1, ..., \tau_n)]\!]).(!\overline{h \cdot \texttt{tup}}\langle v_1, ..., v_n\rangle \mid \overline{o}\langle h\rangle)\right)$$

Then by repeated application of (E-Res1) followed by (E-Com) therefore $q = \tau$ as communication reduction is a τ reduction. This then results in the following process $P = \nu h.(!\overline{h \cdot \texttt{tup}}\langle v_1, ..., v_n\rangle \mid \overline{o}\langle h\rangle)$ and as it was reached using only administrative reductions we have that $[\![(e_1, ..., e_n)]\!]_o^\Gamma \overset{\circ}{\Longrightarrow} P$. Then by going under restriction, and using structural congruence to swap $!\overline{h \cdot \texttt{tup}}\langle v_1, ..., v_n\rangle$ and $\overline{o}\langle h\rangle$ in the parallel composition and then going under it, we have that P can send h on o. Therefore we have that $P \downarrow_{\overline{o}}$.

□

E.6 Proof of Lemma 4.6

Proof of Lemma 4.6.

By inspection of the translation in Chapter 4.1 we have that the constructs $\{x, n, \lambda x : \tau.e, (e_1, ..., e_n), [e_1, ..., e_n]\}$ only contain administrative transitions. Then by induction on $\mathcal{D}(e)$ we get two cases:

$\boldsymbol{\mathcal{D}(e) = 0}$: Then $e \in \{n, x, \lambda x : \tau.e\}$ and by Definition 2.1 we have that $e \in \mathcal{V}$. Then as $e \in \mathcal{V}$ we have by Lemma 4.5 that $[\![e]\!]_o^\Gamma \overset{\circ}{\Longrightarrow} P$ and $P \downarrow_{\overline{o}}$.

$\mathcal{D} > \mathbf{0}$: Then $e \in \{(e_1, ..., e_n), [e_1, ..., e_n]\}$ and $\mathcal{D}(e) = \max_{i \in \{0,...,n\}}(\mathcal{D}(e_i)) + 1$ where $e_i \in \{x, n, \lambda x : \tau.e, (e_1, ..., e_n), [e_1, ..., e_n]\}$. Then as e_i does not contain constructs with important transitions then e must contain fully evaluated expressions and therefore by Definition 2.1 $e \in \mathcal{V}$ and therefore by Lemma 4.5 that $[\![e]\!]_o^\Gamma \overset{\circ}{\Longrightarrow} P$ and $P \downarrow_{\overline{o}}$.

□

E.7 Proof of Theorem 4.1

Proof of Soundness.

We prove this by induction in the rules used for concluding e is well-typed.

(**BT-Var**): From the (BT-Var) rule we know that $e : \tau$. From inspection of the translation we have $\overline{o}\langle x \rangle$ and by the type rules (ET-Send) and (ET-U) we have that $o : \mathrm{ch}(t)$ where $t = \Delta(x)$. Then we have that $[\![\Gamma]\!] = \Delta, \Pi$ and $\Gamma(x) : \tau$ therefore $\Delta(x) = [\![\tau]\!]$ and then it must be that $o : \mathrm{ch}([\![\tau]\!])$.

(**BT-Int**): From the (BT-Int) rule we know that $e : \mathbf{Int}$. From inspection of the translation we have $\overline{o}\langle n \rangle$ and by the type rules (ET-Send) and (ET-N) we have that $o : \mathrm{ch}(t)$ where $t = \mathbf{Int}$. By the translation of types we have $[\![\mathbf{Int}]\!] = \mathbf{Int}$. Therefore $o : \mathrm{ch}([\![\tau]\!])$.

(**BT-Abs**): From the (BT-Abs) rule we know that $e : (\tau_1 \to \tau_2)$. From inspection of the translation we have $\overline{o}\langle h \rangle$ and by the type rule (ET-Send) we have $o : \mathrm{ch}(t)$ where t is the type of the object we are sending on o. From the translation we can see that $h : [\![\tau_1 \to \tau_2]\!]$. Therefore it must be that $o : \mathrm{ch}([\![\tau_1 \to \tau_2]\!])$.

(**BT-App**): From the (BT-App) rule we know that $e : \tau_2$. From inspection of the translation we have an $h : [\![\tau_1 \to \tau_2]\!] = \mathrm{ch}([\![\tau_1]\!], \mathrm{ch}([\![\tau_2]\!]))$. From the translation we see that we send v, o on h and from that we then have $v : [\![\tau_1]\!]$ and $o : \mathrm{ch}([\![\tau_2]\!])$. Therefore soundness hold.

(**BT-Index**): From the (BT-Index) rule we know that $e : \tau$. From inspection of the translation we have $\overline{o}\langle v \rangle$ and by the type rule (ET-Send) we have $o : \mathrm{ch}(t)$ where t is the type of the object we are sending on o. From the translation we can see that $v : [\![\tau]\!]$. Therefore it must be that $o : \mathrm{ch}([\![\tau]\!])$.

(**BT-Bin**): From the (BT-Bin) rule we know that $e : \mathbf{Int}$. From inspection of the translation we have $\overline{o}\langle v_1 \odot v_2 \rangle$ and by the type rule (ET-Send) we have $o : \mathrm{ch}(t)$ where t is the type of the object we are sending on o. From (ET-Bin) we have the type of $v_1 \odot v_2 : \mathbf{Int}$. By the translation of types we have $[\![\mathbf{Int}]\!] = \mathbf{Int}$. Therefore $o : \mathrm{ch}([\![\tau]\!])$.

(**BT-If**): From the (BT-If) rule we know that $e : \tau$ where both sub-expressions e_2 and e_3 is of type τ. From the induction hypothesis we get that from $[\![e_2]\!]_o^\Gamma$ and $[\![e_3]\!]_o^\Gamma$ that $\Delta, \Pi(o) = \mathrm{ch}([\![\tau]\!])$. Therefore soundness hold.

(**BT-Array**): From the (BT-Array) rule we know that $e : [\tau]$. From inspection of the translation we have $\overline{o}\langle h \rangle$ and by the type rule (ET-Send) we have $o : \mathrm{ch}(t)$ where t is the

type of the object we are sending on o. From the translation we can see that $h : [\![[\tau]]\!]$. Therefore it must be that $o : \mathrm{ch}([\![[\tau]]\!])$.

(**BT-Tuple**): From the (BT-Tuple) rule we know that $e : (\tau_1, ..., \tau_n)$. From inspection of the translation we have $\overline{o}\langle h\rangle$ and by the type rule (ET-Send) we have $o : \mathrm{ch}(t)$ where t is the type of the object we are sending on o. From the translation we can see that $h : [\![(\tau_1, ..., \tau_n)]\!]$. Therefore it must be that $o : \mathrm{ch}([\![(\tau_1, ..., \tau_n)]\!])$.

□

Proof of Completeness.

We prove this by induction in the structure of e.

$\boldsymbol{e = x}$: We have that $[\![e]\!]^\Gamma_o = (\Delta, \Pi, P)$ where $P = \overline{o}\langle x\rangle$. We assume that $\Delta(o) = \mathrm{ch}(t)$ where $t = [\![\tau]\!]$, therefore we must have that $\Delta \vdash x : t$. By inspection of the translation of BtF type environment Figure 5.4 we have that $\Delta, x : t$ implies $\Gamma, x : \tau$ and therefore $\Gamma \vdash x : \tau$.

$\boldsymbol{e = n}$: We have that $[\![e]\!]^\Gamma_o = (\Delta, \Pi, P)$ where $P = \overline{o}\langle n\rangle$. We assume that $\Delta(o) = \mathrm{ch}(t)$ where $t = [\![\tau]\!]$, therefore we must have that $\Delta \vdash n : t$ and by (ET-N) we have that $t =$ **Int**. By inspection of the translation of BtF type environment Figure 5.4 and types Figure 5.3 we have that $\Delta, n :$ **Int** implies $\Gamma, n :$ **Int** and therefore $\Gamma \vdash n :$ **Int**.

$\boldsymbol{e = \lambda(x : \tau_1).e_1}$: We have that $[\![e]\!]^\Gamma_o = (\Delta, \Pi, P)$ where

$$P = (\nu h : [\![\tau_1 \to \tau_2]\!]).(\overline{o}\langle h\rangle \mid !h(x : [\![\tau_1]\!], r : \mathrm{ch}([\![\tau_2]\!])).[\![e_1]\!]^\Gamma_r)$$

We assume that $\Delta(o) = \mathrm{ch}(t)$ where $t = [\![\tau]\!]$. By inspection of the translation we have that $\overline{o}\langle h\rangle$ where $h : [\![\tau_1 \to \tau_2]\!]$ which implies $\tau = \tau_1 \to \tau_2$. Then by the induction hypothesis we get that $[\![e_1]\!]^\Gamma_o = (\Delta', \Pi, P')$ where $\Delta' = \Delta, h : [\![\tau_1 \to \tau_2]\!], x : [\![\tau_1]\!], r : \mathrm{ch}([\![\tau_2]\!])$, s.t $\Gamma, x : \tau_1 \vdash e_1 : \tau_2$. Therefore by (BT-Abs) we get that $\Gamma \vdash \lambda(x : \tau_1).e_1 : \tau_1 \to \tau_2$.

$\boldsymbol{e = e_1 e_2}$: We have that $[\![e]\!]^\Gamma_o = (\Delta, \Pi, P)$ where

$$P = (\nu o_1 : \mathrm{ch}([\![\tau_1 \to \tau_2]\!])).(\nu o_2 : \mathrm{ch}([\![\tau_1]\!])).\big([\![e_1]\!]^\Gamma_{o_1} \mid [\![e_2]\!]^\Gamma_{o_2} \mid o_1(h : [\![\tau_1 \to \tau_2]\!]).o_2(v : [\![\tau_1]\!]). \bullet \overline{h}\langle v, o\rangle\big)$$

We assume that $\Delta(o) = \mathrm{ch}(t)$ where $t = [\![\tau]\!]$. By inspection of the translation we have that $\overline{h}\langle v, o\rangle$ where $h : [\![\tau_1 \to \tau_2]\!]$. By inspection of the translation of BtF types Figure 5.3 we have that $h : \mathrm{ch}([\![\tau_1]\!], \mathrm{ch}([\![\tau_2]\!]))$ this then implies that $o : \mathrm{ch}([\![\tau_2]\!])$ and $\tau = \tau_2$. Then by the induction hypothesis we get that $[\![e_1]\!]^\Gamma_{o_1} = (\Delta', \Pi', P')$ where $\Delta(o_1) = \mathrm{ch}([\![\tau_1 \to \tau_2]\!])$ s.t $\Gamma' \vdash e_1 : \tau_1 \to \tau_2$ and $[\![e_2]\!]^\Gamma_{o_2} = (\Delta'', \Pi'', P'')$ where $\Delta(o_2) = \mathrm{ch}([\![\tau_1]\!])$ s.t $\Gamma' \vdash e_2 : \tau_1$. Therefore by (BT-App) we get that $\Gamma \vdash e_1 e_2 : \tau_2$

$\boldsymbol{e = e_1[e_2]}$: We have that $[\![e]\!]^\Gamma_o = (\Delta, \Pi, P)$ where

$$P = (\nu o_1 : \mathrm{ch}([\![[\tau]]\!])).(\nu o_2 : \mathrm{ch}([\![\tau]\!])).[\![e_1]\!]^\Gamma_{o_1} \mid [\![e_2]\!]^\Gamma_{o_2} \mid o_1(h : [\![[\tau]]\!]).$$
$$o_2(i : [\![\mathbf{Int}]\!]).h \cdot i(v : [\![\tau]\!]). \bullet \overline{o}\langle v\rangle$$

We assume that $\Delta(o) = \text{ch}(t)$ where $t = [\![\tau]\!]$. By the induction hypothesis we get that $[\![e_1]\!]^\Gamma_{o_1} = (\Delta_1, \Pi_1, P_1)$ and $[\![e_2]\!]^\Gamma_{o_2} = (\Delta_2, \Pi_2, P_2)$ where $\Delta(o_1) = \text{ch}([\![[\tau]]\!])$ and $\Delta(o_2) = \text{ch}([\![\mathbf{Int}]\!])$ s.t $\Gamma \vdash e_1 : [\tau]$ and $\Gamma \vdash e_2 : \mathbf{Int}$. If P is well-typed we get the following well-typed sub-process $h \cdot i(v : [\![\tau]\!]). \bullet \overline{o}\langle v\rangle$ and thereby we know the type of the object o carries. From this we get that $t = [\![\tau]\!]$ and by (BT-Index) we get that $\Gamma \vdash e_1[e_2] : \tau$.

$\boldsymbol{e = \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3}$: We have that $[\![e]\!]^\Gamma_o = (\Delta, \Pi, P)$ where

$$P = (\nu o_1 : \text{ch}([\![\mathbf{Int}]\!])).\Big([\![e_1]\!]^\Gamma_{o_1} \mid o_1(n : [\![\mathbf{Int}]\!]). \bullet [n \neq 0][\![e_2]\!]^\Gamma_o, [\![e_3]\!]^\Gamma_o\Big)$$

We assume that $\Delta(o) = \text{ch}(t)$ where $t = [\![\tau]\!]$. Then if P is well-typed we also have the following subprocess being well-typed $[n \neq 0][\![e_2]\!]^\Gamma_o, [\![e_3]\!]^\Gamma_o$ in which both branches outputs on the channel o. Then by the induction hypothesis we get that $[\![e_1]\!]^\Gamma_{o_1} = (\Delta_1, \Pi_1, P_1)$ where $\Delta(o_1) = \text{ch}([\![\mathbf{Int}]\!])$ s.t $\Gamma \vdash e_1 : \mathbf{Int}$ and $[\![e_2]\!]^\Gamma_o = (\Delta_2, \Pi_2, P_2)$, $[\![e_3]\!]^\Gamma_o = (\Delta_3, \Pi_3, P_3)$ where $\Delta(o) = \text{ch}([\![\tau]\!])$ s.t $\Gamma \vdash e_2 : \tau$ and $\Gamma \vdash e_3 : \tau$. Therefore by (BT-If) we get that $\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \tau$.

$\boldsymbol{e = e_1 \odot e_2}$: We have that $[\![e]\!]^\Gamma_o = (\Delta, \Pi, P)$ where

$$P = (\nu o_1 : \text{ch}([\![\mathbf{Int}]\!])).(\nu o_2 : \text{ch}([\![\mathbf{Int}]\!])).\Big([\![e_1]\!]^\Gamma_{o_1} \mid [\![e_2]\!]^\Gamma_{o_2} \mid o_1(v_1 : [\![\mathbf{Int}]\!]).$$
$$o_2(v_2 : [\![\mathbf{Int}]\!]). \bullet \overline{o}\langle v_1 \odot v_2\rangle\Big)$$

We assume that $\Delta(o) = \text{ch}(t)$ where $t = [\![\tau]\!]$. By the induction hypothesis we get that $[\![e_1]\!]^\Gamma_{o_1} = (\Delta_1, \Pi_1, P_1)$ and $[\![e_2]\!]^\Gamma_{o_2} = (\Delta_2, \Pi_2, P_2)$ where $\Delta(o_1) = \text{ch}([\![\mathbf{Int}]\!])$ and $\Delta(o_2) = \text{ch}([\![\mathbf{Int}]\!])$ s.t $\Gamma \vdash e_1 : \mathbf{Int}$ and $\Gamma \vdash e_2 : \mathbf{Int}$. Then by inspection of the translation we have that $\overline{o}\langle v_1 \odot v_2\rangle$ and by (ET-Bin) we have that $\Delta, v_1 : \mathbf{Int}, v_2 : \mathbf{Int}, \Pi \vdash v_1 \odot v_2 : \mathbf{Int}$ which implies $\tau = \mathbf{Int}$. Therefore by (BT-Bin) we get that $\Gamma \vdash e_1 \odot e_2 : \mathbf{Int}$.

$\boldsymbol{e = [e_1, ..., e_n]}$: We have that $[\![e]\!]^\Gamma_o = (\Delta, \Pi, P)$ where

$$P = (\nu o_1 : \text{ch}([\![\tau]\!])).....(\nu o_n : \text{ch}([\![\tau]\!])).(\nu h : [\![[\tau]]\!]).$$
$$\left(\prod_{i=1}^{n} [\![e_i]\!]^\Gamma_{o_i} \mid o_1(v_1 : [\![\tau]\!]).....o_n(v_n : [\![\tau]\!]).\right.$$
$$\left.\left(\prod_{i=1}^{n} Cell(h, i-1, v_i, [\![\tau]\!]) \mid \overline{h \cdot \texttt{len}}\langle n\rangle \mid \overline{o}\langle h\rangle\right)\right)$$

We assume that $\Delta(o) = \text{ch}(t)$ where $t = [\![\tau]\!]$. By the induction hypothesis we get that $\forall i \in 1..n.[\![e_i]\!]^\Gamma_{o_i} = (\Delta_i, \Pi_i, P_i)$ where $\Delta(o_i) = \text{ch}([\![\tau_1]\!])$ such that $\Gamma \vdash e_i : \tau_1$. If P is well-typed the following sub-process in the translation must be well-typed as well: $\overline{o}\langle h\rangle$. From the restriction $(\nu h : [\![[\tau_1]]\!])$ we get the type of the object that o carries. From this we get that $t = [\![[\tau_1]]\!]$. We must therefore have that $\Gamma \vdash e : [\tau_1]$ and by (BT-Array) we get that $\Gamma \vdash [e_1, ..., e_n] : [\tau_1]$.

$\boldsymbol{e = (e_1, ..., e_n)}$: We have that $[\![e]\!]^\Gamma_o = (\Delta, \Pi, P)$ where

$$P = (\nu o_1 : \mathrm{ch}([\![\tau_1]\!])).....(\nu o_n : \mathrm{ch}([\![\tau_n]\!])).$$
$$\Bigg(\prod_{i=1}^{n} [\![e_i]\!]^{\Gamma}_{o_i} \mid o_1(v_1 : [\![\tau_1]\!]).\ ...\ .o_n(v_n : [\![\tau_n]\!]).$$
$$\nu(h : [\![(\tau_1, ..., \tau_n)]\!]).\big(!\overline{h \cdot \texttt{tup}}\langle v_1, ..., v_n\rangle \mid \overline{o}\langle h\rangle\big)\Bigg)$$

We assume that $\Delta(o) = \mathrm{ch}(t)$ where $t = [\![\tau]\!]$. By the induction hypothesis we get that $\forall i \in 1..n.[\![e_i]\!]^{\Gamma}_{o_i} = (\Delta_i, \Pi_i, P_i)$ where $\Delta(o_i) = \mathrm{ch}([\![\tau_i]\!])$ such that $\Gamma \vdash e_i : \tau_i$. If P is well-typed the following sub-process in the translation must also be well-typed: $\overline{o}\langle h\rangle$. From the restriction $(\nu h : [\![(\tau_1, ..., \tau_n)]\!])$ we get the type of the object that o carries. From this we get that $t = [\![(\tau_1, ..., \tau_n)]\!]$. We must therefore have that $\Gamma \vdash e : (\tau_1, ..., \tau_n)$ and by (BT-Tuple) we get that $\Gamma \vdash (e_1, ..., e_n) : (\tau_1, ..., \tau_n)$.

$\boldsymbol{e = \texttt{size}\ e_1}$: We have that $[\![e]\!]^{\Gamma}_{o} = (\Delta, \Pi, P)$ where

$$P = (\nu o_1 : \mathrm{ch}([\![[\tau_1]]\!])).\big([\![e_1]\!]^{\Gamma}_{o_1} \mid$$
$$o_1(h : [\![[\tau_1]]\!]).h \cdot \texttt{len}\ (n : \mathbf{Int}).\bullet\, \overline{o}\langle n\rangle\big)$$

We assume that $\Delta(o) = \mathrm{ch}(t)$ where $t = [\![\tau]\!]$. By the induction hypothesis we get that $[\![e_1]\!]^{\Gamma}_{o_1} = (\Delta_1, \Pi_1, P_1)$ where $\Delta(o_1) = \mathrm{ch}([\![[\tau_1]]\!])$ s.t $\Gamma \vdash e_1 : [\tau_1]$. Then if P is well-typed the following sub-process from the translation must be well-typed: $h \cdot \texttt{len}\ (n : \mathbf{Int}).\bullet\ \overline{o}\langle n\rangle)$. From this we know the type of object that o carries from which we get that $t = \mathbf{Int}$ and therefore we must have that $\Gamma \vdash e : \mathbf{Int}$. From Definition 3.3 we get the type of $\texttt{size} : [\tau_1] \to \mathbf{Int}$ and by (BT-App) we get that $\Gamma \vdash \texttt{size}\ e_1 : \mathbf{Int}$.

$\boldsymbol{e = \texttt{iota}\ e_1}$: We have that $[\![e]\!]^{\Gamma}_{o} = (\Delta, \Pi, P)$ where

$$P = (\nu o_1 : \mathrm{ch}([\![\mathbf{Int}]\!])).(\nu r : \mathrm{ch}(\mathbf{Int})).(\nu d : \mathrm{ch}(\mathbf{Int})).$$
$$(\nu h : @\{\mathrm{ch}(\mathbf{Int}, \mathbf{Int}), \mathrm{ch}(\mathbf{Int}), \mathrm{ch}(\mathbf{Int})\}).$$
$$\big([\![e_1]\!]^{\Gamma}_{o_1} \mid o_1(n : [\![\mathbf{Int}]\!]).\big(\mathit{Repeat}(n, r, d) \mid$$
$$!r(i : \mathbf{Int}).\ \mathit{Cell}(h, i, i) \mid \bullet\, d(\_ : \mathbf{Int}).$$
$$\overline{o}\langle h\rangle \mid \overline{h.\ \texttt{len}}\langle n : \mathbf{Int}\rangle\big)\big)$$

We assume that $\Delta(o) = \mathrm{ch}(t)$ where $t = [\![\tau]\!]$. By the induction hypothesis we that $[\![e_1]\!]^{\Gamma}_{o_1} = (\Delta_1, \Pi_1, P_1)$ where $\Delta(o_1) = \mathrm{ch}(\mathbf{Int})$ such that $\Gamma \vdash e_1 : \mathbf{Int}$. Then if P is well-typed the following sub-process is also well-typed: $\overline{o}\langle h\rangle$. From the restriction $(\nu h : @\{\mathrm{ch}(\mathbf{Int}, \mathbf{Int}), \mathrm{ch}(\mathbf{Int}), \mathrm{ch}(\mathbf{Int})\})$ which is the type of the object that o carries. From the translation of types know that $@\{\mathrm{ch}(\mathbf{Int}, \mathbf{Int}), \mathrm{ch}(\mathbf{Int}), \mathrm{ch}(\mathbf{Int})\} = [\![[\mathbf{Int}]]\!]$ and we must therefore have $\Gamma \vdash e : [\mathbf{Int}]$. From Definition 3.3 we get the type of $\texttt{iota} : \mathbf{Int} \to [\mathbf{Int}]$ and by (BT-App) we get that $\Gamma \vdash \texttt{iota}\ e_1 : [\mathbf{Int}]$.

$\boldsymbol{e = \texttt{map}\ e_1}$: We have that $[\![e]\!]^{\Gamma}_{o} = (\Delta, \Pi, P)$ where

$$P = (\nu o_1 : \mathrm{ch}([\![(\tau_1 \to \tau_2, [\tau_1])]\!])).(\nu h_1 : [\![[\tau_2]]\!]).\Big([\![e_1]\!]_{o_1} \mid$$
$$o_1(h_2 : [\![(\tau_1 \to \tau_2, [\tau_1])]\!]).h_2 \cdot \texttt{tup}\ (f : [\![\tau_1 \to \tau_2]\!], h_3 : [\![[\tau_1]]\!]).$$
$$h_3 \cdot \texttt{len}\ (n : \mathbf{Int}).(\nu h_4 : \mathrm{ch}(\mathbf{Int}, [\![\tau_1]\!])).$$
$$\overline{h_3 \cdot \texttt{all}}\langle h_4\rangle.(\nu r_1 : \mathrm{ch}(\mathbf{Int})).(\nu d : \mathrm{ch}(\mathbf{Int})).$$
$$\Big(\mathit{Repeat}(n, r_1, d) \mid !h_4(i : \mathbf{Int}, v_1 : [\![\tau_1]\!]).$$
$$(\nu r_2 : \mathrm{ch}([\![\tau_2]\!])).\overline{f}\langle v_1, r_2\rangle.r_2(v_2 : [\![\tau_2]\!])$$
$$.r_1(\_ : \mathbf{Int}).\ \mathit{Cell}(h_1, i, v_2, [\![\tau_2]\!]) \mid$$
$$\bullet\, d(\_ : \mathbf{Int}).\overline{o}\langle h_1\rangle \mid !\overline{h_1 \cdot \texttt{len}}\langle n\rangle\Big)\Big)$$

We assume that $\Delta(o) = \mathrm{ch}(t)$ where $t = [\![\tau]\!]$. By the induction hypothesis we that $[\![e_1]\!]^\Gamma_{o_1} = (\Delta_1, \Pi_1, P_1)$ where $\Delta(o_1) = \mathrm{ch}([\![(\tau_1 \to \tau_2, [\tau_1])]\!])$ such that $\Gamma \vdash e_1 : (\tau_1 \to \tau_2, [\tau_1])$. Then if P is well-typed the following sub-process must well-typed: $\overline{o}\langle h_1\rangle$. From the restriction we get $(\nu h_1 : [\![[\tau_2]]\!])$ which is the type of the object that o carries and therefore we must have that $\Gamma \vdash e : [\tau_2]$. From Definition 3.3 we get the type of $\texttt{map} : (\tau_1 \to \tau_2, [\tau_1]) \to [\tau_2]$ and by (BT-App) we get that $\Gamma \vdash \texttt{map}\ e_1 : [\tau_2]$.

□

E.8 Proof of Theorem 4.2

Proof of Theorem 4.2.

Let $\mathcal{B}$ be the set of all BtF programs and let R be the following relation $R = \big\{(e, [\![e]\!]^\Gamma_o) \mid e \in \mathcal{B}, o \text{ fresh}\big\}$. We show that R is an administrative operational correspondences. As per Definition 4.4 we only consider pairs where $e \to e'$ and where $[\![e]\!]^\Gamma_o$ contains $\bullet$.

Array: For arrays we have $e = [e_1, ..., e_n]$ and must prove that the two parts of operational correspondence hold.

Soundness: For the first part we have that if $e \to e'$ then there is a P' such that $[\![e]\!]^\Gamma_o \overset{\bullet}{\Longrightarrow} P'$ and $P' \dot{\approx}_\alpha [\![e']\!]^\Gamma_o$. From the rule (B-Array) we know there must exist an i such that $e_i \to e_i'$. The translation $[\![e]\!]^\Gamma_o$ contains $(\nu o_i : t_i).\Big([\![e_i]\!]^\Gamma_{o_i}\Big)$. By our assumption that $(e, [\![e]\!]^\Gamma_o) \in R$ we have that $\Big(e_i, [\![e_i]\!]^\Gamma_{o_i}\Big) \in R$. Then it follows that we have $[\![e_i]\!]^\Gamma_{o_i} \overset{\bullet}{\Longrightarrow} Q$ and that $Q \dot{\approx}_\alpha [\![e_i']\!]^\Gamma_{o_i}$.

We let the subprocess $[\![e_i]\!]^\Gamma_{o_i}$ in $[\![e]\!]^\Gamma_o$ be replaced by Q. We then have that when $[\![e_i]\!]^\Gamma_{o_i}$ is unguarded we know that $[\![e]\!]^\Gamma_o \overset{\bullet}{\Longrightarrow} P'$ and $P' \dot{\approx}_\alpha [\![e']\!]^\Gamma_o$.

Completeness: If $[\![e]\!]^\Gamma_o \overset{\bullet}{\Longrightarrow} P'$, then $P' \dot{\approx}_\alpha [\![e']\!]^\Gamma_o$ where $e \to e'$. However as the translation of (B-Array) on page 27 contains no $\bullet$ then the important reduction must be in one of the elements of e. Then by our assumption that $[\![e]\!]^\Gamma_o \overset{\bullet}{\Longrightarrow} P'$ there must exist an i s.t in the sub-process $[\![e_i]\!]^\Gamma_{o_i}$ the important reduction occurs. As $\Big(e_i, [\![e_i]\!]^\Gamma_{o_i}\Big) \in R$ we know that $[\![e_i]\!]^\Gamma_{o_i} \overset{\bullet}{\Longrightarrow} Q$ for any Q and e_i' where $Q \dot{\approx}_\alpha [\![e_i]\!]^\Gamma_{o_i}$ and $e_i \to e_i'$.

Then we select e' to be e where e_i is replaced by e_i'. Therefore $P' \dot{\approx}_\alpha [\![e']\!]_o^\Gamma$ as only administrative transitions has been taken.

Tuple: For tuples we have $e = (e_1, ..., e_n)$ and must prove that the two parts of operational correspondence hold.

Soundness: For the first part we have that if $e \to e'$ then there is a P' such that $[\![e]\!]_o^\Gamma \overset{\bullet}{\Longrightarrow} P'$ and $P' \dot{\approx}_\alpha [\![e']\!]_o^\Gamma$. From the rule (B-Tuple) we know there must exist an i such that $e_i \to e_i'$. The translation $[\![e]\!]_o^\Gamma$ contains $(\nu o_i : t_i).\left([\![e_i]\!]_{o_i}^\Gamma\right)$. By our assumption that $(e, [\![e]\!]_o^\Gamma) \in R$ we have that $\left(e_i, [\![e_i]\!]_{o_i}^\Gamma\right) \in R$. Then it follows that we have $[\![e_i]\!]_{o_i}^\Gamma \overset{\bullet}{\Longrightarrow} Q$ and that $Q \dot{\approx}_\alpha [\![e_i']\!]_{o_i}^\Gamma$.

We let the subprocess $[\![e_i]\!]_{o_i}^\Gamma$ in $[\![e]\!]_o^\Gamma$ be replaced by Q. We then have that when $[\![e_i]\!]_{o_i}^\Gamma$ is unguarded we know that $[\![e]\!]_o^\Gamma \overset{\bullet}{\Longrightarrow} P'$ and $P' \dot{\approx}_\alpha [\![e']\!]_o^\Gamma$.

Completeness: For the second part we have that if $[\![e]\!]_o^\Gamma \overset{\bullet}{\Longrightarrow} P'$ then there is an e' such that $e \to e'$ and $P' \dot{\approx}_\alpha [\![e']\!]_o^\Gamma$. However as the translation of (B-Tuple) on page 27 contains no $\bullet$ then the important reduction must be in one of the elements of e. Then by our assumption that $[\![e]\!]_o^\Gamma \overset{\bullet}{\Longrightarrow} P'$ there must exist an i such that in the subprocess $[\![e_i]\!]_{o_i}^\Gamma$ the important reduction occurs. As we have that $\left(e_i, [\![e_i]\!]_{o_i}^\Gamma\right) \in R$ we know that $[\![e_i]\!]_{o_i}^\Gamma \overset{\bullet}{\Longrightarrow} Q$ for any Q and e_i' where $Q \dot{\approx}_\alpha [\![e_i]\!]_{o_i}^\Gamma$ and $e_i \to e_i'$.

Then we select e' to be e where e_i is replaced by e_i'. Therefore we have $P' \dot{\approx}_\alpha [\![e']\!]_o^\Gamma$ as only administrative transitions has been taken.

Indexing: For indexing we have $e = e_1[e_2]$ and must prove that the two parts of operational correspondence hold.

Soundness: For the first part we have that if $e \to e'$ then there is a P' such that $[\![e]\!]_o^\Gamma \overset{\bullet}{\Longrightarrow} P'$ and $P' \dot{\approx}_\alpha [\![e']\!]_o^\Gamma$. For indexing we have three rules for e to take a step: (B-Index), (B-Index1) and (B-Index2). The two rules, (B-Index1) and (B-Index2), are used for evaluating each sub-expression. By our assumption that $(e, [\![e]\!]_o^\Gamma) \in R$ we have that $\left(e_1, [\![e_1]\!]_{o_1}^\Gamma\right) \in R$, and then if $e_1[e_2] \to e_1'[e_2]$ we have $[\![e_1]\!]_{o_1}^\Gamma \overset{\bullet}{\Longrightarrow} [\![e_1']\!]_{o_1}^\Gamma$ and $[\![e_1]\!]_{o_1}^\Gamma \dot{\approx}_\alpha [\![e_1']\!]_{o_1}^\Gamma$. We have that when $[\![e_1]\!]_{o_1}^\Gamma$ is unguarded we know that $[\![e]\!]_o^\Gamma \overset{\bullet}{\Longrightarrow} P'$ and $P' \dot{\approx}_\alpha [\![e']\!]_o^\Gamma$. The same holds for e_2.

For the last rule (B-Index) we have that if $v_1[v_2] \to v_3$ then a corresponding operation exists. By (B-Index) we have v_1 being an array of size n and v_2 being a integer with a value of at most n. By Lemma 4.5 we have that the translation of two expression have observable action on their respective o channel and these are administrative reductions. Then from the third case of Lemma 4.1 we have $(\nu h : t_h).(Q_h \mid h \cdot i(v : t).\bullet\,\overline{o}\langle v\rangle, \mathbf{0})$ where $t_h = (i : \mathrm{ch}(t_1), \mathtt{all} : \mathrm{ch}(t_1), \mathtt{len} : \mathrm{ch}(\mathbf{Int}))$ with $t_1 = (i : \mathbf{Int}, v : t)$ and Q_h is the leftovers from the reduced array $([\![e_1]\!]_{o_1}^\Gamma)$ and index $([\![e_2]\!]_{o_2}^\Gamma)$. Using Lemma 4.4 we can remove Q_h and then we have $[\![e]\!]_o^\Gamma \overset{\bullet}{\Longrightarrow} P'$ and $[\![e']\!]_o^\Gamma \dot{\approx}_\alpha P'$.

Completeness: If $[\![e]\!]_o^\Gamma \overset{\bullet}{\Longrightarrow} P'$, then $P' \dot{\approx}_\alpha [\![e']\!]_o^\Gamma$ where $e \to e'$. From the translation of (B-Index) in Chapter 4.1.1.1 there are three possible locations for important

reductions $[\![e_1]\!]^\Gamma_{o_1}, [\![e_2]\!]^\Gamma_{o_2}$ and by the index check. Therefore we get two sub-cases one for where the important reduction is in the sub-processes and one where it is before the output of the value.

Then because $[\![e]\!]^\Gamma_o \overset{\bullet}{\Longrightarrow} P'$ there must exist an i where $i \in \{1, 2\}$ s.t in the sub-process $[\![e_i]\!]^\Gamma_{o_i}$ the important reduction occurs. We assume that exists some $\left(e_j, [\![e_j]\!]^\Gamma_{o_j}\right) \in R$ and from that assumption we have that $[\![e_j]\!]^\Gamma_{o_j} \overset{\bullet}{\Longrightarrow} Q$ for any Q and e'_j where $Q \dot{\approx}_\alpha [\![e_i]\!]^\Gamma_{o_i}$ and $e_j \to e'_j$. Then we select e' to be e where e_i is replaced by e'_i. Therefore $P \dot{\approx}_\alpha [\![e']\!]^\Gamma_o$ as only administrative transitions has been taken.

Then because $[\![e]\!]^\Gamma_o \overset{\bullet}{\Longrightarrow} P'$ there must be a process listening on $h \cdot i$ and this is the only case when the array and index is fully reduced. We have that we can only listen on $h \cdot i$ if the array size is larger or equal to i. Therefore $e \to e'$ by (B-Index) and $P' \dot{\approx}_\alpha [\![e']\!]^\Gamma_o$.

Binop: For Binop we have $e = e_1 \odot e_2$ and must prove that the two parts of operational correspondence hold.

Soundness: For the first part we have that if $e \to e'$ then there is a P' such that $[\![e]\!]^\Gamma_o \overset{\bullet}{\Longrightarrow} P'$ and $[\![e']\!]^\Gamma_o \dot{\approx}_\alpha P'$.

Just like indexing we have three rules for e to take a step: (B-Bin), (B-Bin1) and (B-Bin2). The two rules, (B-Bin1) and (B-Bin2), are used for evaluating each sub-expression. By our assumption that $(e, [\![e]\!]^\Gamma_o) \in R$ we have that $\left(e_1, [\![e_1]\!]^\Gamma_{o_1}\right) \in R$, and then if $e_1 \odot e_2 \to e'_1 \odot e_2$ we have $[\![e_1]\!]^\Gamma_{o_1} \overset{\bullet}{\Longrightarrow} [\![e'_1]\!]^\Gamma_{o_1}$ and $[\![e_1]\!]^\Gamma_{o_1} \dot{\approx}_\alpha [\![e'_1]\!]^\Gamma_{o_1}$. We have that when $[\![e_1]\!]^\Gamma_{o_1}$ is unguarded we know that $[\![e]\!]^\Gamma_o \overset{\bullet}{\Longrightarrow} P'$ and $P' \dot{\approx}_\alpha [\![e']\!]^\Gamma_o$. The same holds for e_2.

For the last rule (B-Bin) we have that if $v_1 \odot v_2 \to v_3$. By Lemma 4.5 we have that the two sub-expressions will output on their respective channel after som administrative reductions. On the important reduction we have that $\overset{\bullet}{\longrightarrow} \overline{o}\langle v_1 \odot v_2 \rangle$ and thereby we have $[\![e]\!]^\Gamma_o \overset{\bullet}{\Longrightarrow} P'$ and $[\![e']\!]^\Gamma_o \dot{\approx}_\alpha P'$.

Completeness: For the second part we have that if $[\![e]\!]^\Gamma_o \overset{\bullet}{\Longrightarrow} P'$ then there is an e' such that $e \to e'$ and $P' \dot{\approx}_\alpha [\![e']\!]^\Gamma_o$. We have three cases: one where the important reduction happens in $[\![e_1]\!]^\Gamma_{o_1}$, one where it happens in $[\![e_2]\!]^\Gamma_{o_2}$ or where it happens on the communication on o.

First case the $\overset{\bullet}{\Longrightarrow}$ transition happens inside $[\![e_1]\!]^\Gamma_{o_1}$. We let $e' = e'_1 \odot e_2$ where $e_1 \to e'_1$. By our assumption that $(e, [\![e]\!]^\Gamma_o \in R)$ we have that $\left(e_1, [\![e_1]\!]^\Gamma_{o_1}\right) \in R$ which means we know that $P' \dot{\approx}_\alpha [\![e']\!]^\Gamma_o$. The same argument follows for the second case where $\overset{\bullet}{\Longrightarrow}$ happens in $[\![e_2]\!]^\Gamma_{o_2}$.

In the last case we have that we receive on o_1 and o_2 and therefore by Lemma 4.6 we know that both e_1 and e_2 must be values. We then select $e' = v_3$ and then we have $P' \dot{\approx}_\alpha [\![e']\!]^\Gamma_o$.

Application: For application we have $e = e_1 e_2$ and must prove that the two parts of operational correspondence hold.

Soundness: For the first part we have that if $e \to e'$ then there is a P' such that $[\![e]\!]_o^\Gamma \overset{\bullet}{\Longrightarrow} P'$ and $P' \dot{\approx}_\alpha [\![e']\!]_o^\Gamma$. There are two cases for when $e \to e'$: One when the sub-expressions can take a step and one when they cannot.

For the first case we have that there are two application rules for the sub-expressions e_1 and e_2 to take a step: (B-App1) and (B-App2). By our assumption that $(e, [\![e]\!]_o^\Gamma) \in R$ we have that $(e_1, [\![e_1]\!]_o^\Gamma) \in R$. The same holds for e_2. When $[\![e_1]\!]_{o_1}^\Gamma$ and $[\![e_2]\!]_{o_2}^\Gamma$ is unguarded we know that $[\![e]\!]_o^\Gamma \overset{\bullet}{\Longrightarrow} P'$ and $P' \dot{\approx}_\alpha [\![e']\!]_o^\Gamma$.

For the second case we have that neither e_1 and e_2 cannot take a step and by Lemma 4.6 we know that e_1 and e_2 must be values and therefore must be on the form $e_1 = \lambda(x : \tau_1).e_\lambda$ and $e_2 = v$. In this case we can take a step with (B-Abs) and this can be matched in the translation of application. As this case is more complicated we will show how the translation will match this. First the two sub-expression is evaluated and this will give us the process on the following form:

$$(\nu o_1 : \mathrm{ch}([\![\tau_1 \to \tau_2]\!])).(\nu o_2 : \mathrm{ch}([\![\tau_1]\!])).$$
$$\Bigg(\underbrace{(\nu h : [\![\tau_1 \to \tau_2]\!]).(\overline{o}\langle h\rangle \mid\ !h(x : [\![\tau_1]\!], r : \mathrm{ch}([\![\tau_2]\!])).[\![e_\lambda]\!]_r^\Gamma)}_{[\![e_1]\!]_{o_1}^\Gamma}$$
$$\mid \underbrace{(\nu v : [\![\tau_1]\!]).\overline{o_2}\langle v\rangle \mid S}_{[\![e_2]\!]_{o_2}^\Gamma} \mid o_1(h : [\![\tau_1 \to \tau_2]\!]).o_2(v : [\![\tau_1]\!]).\bullet\, \overline{h}\langle v, o\rangle \Bigg)$$

As the translation of e_1 is an abstraction it is substituted with the translation of abstraction. The translation of e_2 is substituted with a value ready to be sent on o_2 in parallel with a processes S that maintains the value. To not confuse the reader, the expression in the translation of abstraction has been renamed to e_λ. After communication on o_1 and o_2 happens the application will be on the following form:

$$(\nu h : [\![\tau_1 \to \tau_2]\!]).(\nu v : [\![\tau_1]\!]).\underbrace{!h(x : [\![\tau_1]\!], r : \mathrm{ch}([\![\tau_2]\!])).[\![e_\lambda]\!]_r^\Gamma}_{\text{Abstraction}} \mid S \mid \bullet\, \overline{h}\langle v, o\rangle$$

Now we can send on the function handle h and thus proceed to $[\![e_\lambda]\!]_r^\Gamma$ where r is the return channel substituted with the output channel o and value v. We denote this as the process H which then corresponds to $H = [\![e_\lambda]\!]_o^\Gamma\{^v/_x\}$. By Lemma 4.2 we have that this corresponds to $e_\lambda\{v \mapsto x\}$ which is our e'. Thereby we have the $[\![e]\!]_o^\Gamma \overset{\bullet}{\Longrightarrow} P'$ and $P' \dot{\approx}_\alpha [\![e']\!]_o^\Gamma$.

Completeness: For the second part we have that if $[\![e]\!]_o^\Gamma \overset{\bullet}{\Longrightarrow} P'$ then there is an e' such that $e \to e'$ and $P' \dot{\approx}_\alpha [\![e']\!]_o^\Gamma$. We have two cases: first if the important transition happens in either $[\![e_1]\!]_{o_1}^\Gamma$ or $[\![e_2]\!]_{o_2}^\Gamma$, or second when sending on the handle h.

In the first case we can select e' to be either $e_1' e_2$ or $e_1 e_2'$ depending on where the important transition happens and then by one of the two application rules we have $e \to e'$.

In the second case both $[\![e_1]\!]^\Gamma_{o_1}$ and $[\![e_2]\!]^\Gamma_{o_2}$ can send on their respective o after some administrative reductions. By Lemma 4.6 we know that e_1 and e_2 are values and as such no important transition exists in those. We know that $[\![e_1]\!]^\Gamma_{o_1}$ is an abstraction and therefore by (B-Abs) we have that $e \to e'$.

Conditional: For conditional we have $e = \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$ and must prove that the two parts of operational correspondence hold.

Soundness: For the first part we have that if $e \to e'$ then there is a P' such that $[\![e]\!]^\Gamma_o \overset{\bullet}{\Longrightarrow} P'$ and $[\![e']\!]^\Gamma_o \mathrel{\dot{\approx}}_\alpha P'$. We have three rules for conditionals: (B-If), (B-Ift) and (B-Iff).

We start by looking at the case for evaluating e_1. By our assumption that $(e, [\![e]\!]^\Gamma_o) \in R$ we have that $\left(e_1, [\![e_1]\!]^\Gamma_{o_1}\right) \in R$ and then if $\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \to \texttt{if}\ e_1'\ \texttt{then}\ e_2\ \texttt{else}\ e_3$ by (B-If) we have $[\![e_1]\!]^\Gamma_{o_1} \overset{\bullet}{\Longrightarrow} [\![e_1']\!]^\Gamma_{o_1}$ and $[\![e_1]\!]^\Gamma_{o_1} \mathrel{\dot{\approx}}_\alpha [\![e_1']\!]^\Gamma_{o_1}$. We then have that when $[\![e_1]\!]^\Gamma_{o_1}$ is unguarded we know that $[\![e]\!]^\Gamma_o \overset{\bullet}{\Longrightarrow} P'$ and $P' \mathrel{\dot{\approx}}_\alpha [\![e']\!]^\Gamma_o$.

When e_1 has evaluated and a value has been sent on o_1 we have one of two possible reductions left. Since $e_1 \not\to$ we know by Lemma 4.6 that e_1 is a value and then we have either $[n \neq 0].[\![e_2]\!]^\Gamma_o, [\![e_3]\!]^\Gamma_o \overset{\bullet}{\longrightarrow} [\![e_2]\!]^\Gamma_o$ (which can be matched using (B-Ift)) or $\overset{\bullet}{\longrightarrow} [\![e_3]\!]^\Gamma_o$ (which can be matched using (B-Iff)). We then have $[\![e]\!]^\Gamma_o \overset{\bullet}{\Longrightarrow} P'$ $P' \mathrel{\dot{\approx}}_\alpha [\![e']\!]^\Gamma_o$.

Completeness: For the second part we have that if $[\![e]\!]^\Gamma_o \overset{\bullet}{\Longrightarrow} P'$ then there is a e' such that $e \to e'$ and $P' \mathrel{\dot{\approx}}_\alpha [\![e']\!]^\Gamma_o$. We have two cases to look at.

First case the $\overset{\bullet}{\Longrightarrow}$ transition happens inside $[\![e_1]\!]^\Gamma_{o_1}$. We let $e' = \texttt{if}\ e_1'\ \texttt{then}\ e_2\ \texttt{else}\ e_3$ where $e_1 \to e_1'$. By our assumption that $(e, [\![e]\!]^\Gamma_o \in R)$ we have that $\left(e_1, [\![e_1]\!]^\Gamma_{o_1}\right) \in R$ which means we know that $P' \mathrel{\dot{\approx}}_\alpha [\![e']\!]^\Gamma_o$.

For the second case we have received values on o_1 and therefore by Lemma 4.6 we know that e_1 must be a value. We can then select e' to be either e_2 or e_3. Depending on how the match concludes we have that either $\overset{\bullet}{\longrightarrow} [\![e_2]\!]^\Gamma_{o_2}$ or $\overset{\bullet}{\longrightarrow} [\![e_3]\!]^\Gamma_o$ and given our assumption that $(e, [\![e]\!]^\Gamma_o \in R)$ we have that $\{(e_2, [\![e_2]\!]^\Gamma_o), (e_3, [\![e_3]\!]^\Gamma_o)\} \subseteq R$ and then we have $P' \mathrel{\dot{\approx}}_\alpha [\![e']\!]^\Gamma_o$.

Map: For map we have $e = \texttt{map}\ e_1$ and must prove that the two parts of operational correspondence hold.

Soundness: For the first part we have that if $e \to e'$ then there is a P' such that $[\![e]\!]^\Gamma_o \overset{\bullet}{\Longrightarrow} P'$ and $P' \mathrel{\dot{\approx}}_\alpha [\![e']\!]^\Gamma_o$.

For the first case we have that e_1 has not yet been evaluated to a tuple. We can then use (B-App2) to take a step.

$$\text{(B-App2)}\ \frac{e_2 \to e_2'}{e_1 e_2 \to e_1 e_2'}$$

As `map` is a variation of application we can use (B-App2). By our assumption that $(e, [\![e]\!]_o^\Gamma) \in R$ we have that $\left(e_1, [\![e_1]\!]_{o_1}^\Gamma\right) \in R$. When $[\![e_1]\!]_{o_1}^\Gamma$ is unguarded we know that $[\![e]\!]_o^\Gamma \overset{\bullet}{\Longrightarrow} P'$ and $P' \dot{\approx}_\alpha [\![e']\!]_o^\Gamma$.

For the second case we have that by (B-Map) we can take a step and therefore e_1 is the tuple $(\lambda x.e_\lambda, [v_1, ..., v_n])$. As this is one of the more complicated cases we will show how this is matched in the translation. First have that $e' = [e_\lambda\{x \mapsto v_1\}, ..., e_\lambda\{x \mapsto v_n\}]$ from (B-Map). From the translation we have:

$$(\nu o_1 : \mathrm{ch}(t_1)).(\nu h_1 : [\![[\tau_2]]\!].[\![e_1]\!]_{o_1}^\Gamma \mid o_1(h_2 : t_1).\underbrace{h_2 \cdot \texttt{tup}\ (f : [\![\tau_1 \to \tau_2]\!], h_3 : [\![[\tau_1]]\!])}_{(\lambda x.e_\lambda, [v_1,...,v_n])}.$$

Here we receive the tuple after the unguarded $[\![e_1]\!]_{o_1}^\Gamma$. Only administrative reductions has happened so far. Next we receive the length of the input array and the values located at each index.

$$\underbrace{h_3 \cdot \texttt{len}\ (n : \mathbf{Int})}_{\text{Input array length}}.\underbrace{(\nu h_4 : \mathrm{ch}(\mathbf{Int}, [\![\tau_1]\!])).\overline{h_3 \cdot \texttt{all}}\langle h_4\rangle}_{\forall (n,v) \in [v_1,...,v_n]}.(\nu r_1 : \mathrm{ch}(\mathbf{Int})).(\nu d : \mathrm{ch}(\mathbf{Int})).$$

Next, using Repeat, we apply the abstraction to each value and pass it to the Cell process to create the new array. Again only administrative reduction has happened. Only when the Repeat process has finished will we receive on d which is an important reduction.

$$\Bigg(\mathit{Repeat}(n, r_1, d) \mid \underbrace{!h_4(i : \mathbf{Int}, v_1 : [\![\tau_1]\!])}_{\text{Index \& argument}}.\underbrace{(\nu r_2 : \mathrm{ch}([\![\tau_2]\!])).\overline{f}\langle v_1, r_2\rangle.r_2(v_2 : [\![\tau_2]\!])}_{\lambda x.e_\lambda v_i}$$

$$.r_1(\_ : \mathbf{Int}).\ \mathit{Cell}(h_1, i, v_2, [\![\tau_2]\!]) \mid \bullet\, d(\_ : \mathbf{Int}).\overline{o}\langle h_1\rangle \mid !\overline{h_1 \cdot \texttt{len}}\langle n\rangle \Bigg)$$

This is our P' and as we output the new updated array we have that $P' \dot{\approx}_\alpha [\![e']\!]_o^\Gamma$.

Completeness: For the second part we have that if $[\![e]\!]_o^\Gamma \overset{\bullet}{\Longrightarrow} P'$ then there is a e' such that $e \to e'$ and $P' \dot{\approx}_\alpha [\![e']\!]_o^\Gamma$.

We know that P' has received on $\bullet\, d(\_ : \mathbf{Int})$, and as we are well-typed we know that $[\![e_1]\!]_{o_1}^\Gamma$ where

$$o_1 : \mathrm{ch}(@\{\mathrm{ch}(\mathrm{ch}(t_1, \mathrm{ch}(t_2)), @\{\mathrm{ch}(\mathbf{Int}, t_1), \mathrm{ch}(t_1), \mathrm{ch}(\mathbf{Int})\})\} = [\![((\tau_1 \to \tau_2), [\tau_1])]\!])$$

Then on P' we observe $P' \downarrow_o$ where $o : \mathrm{ch}(@\{\mathrm{ch}(\mathbf{Int}, t_2), \mathrm{ch}(t_2), \mathrm{ch}(\mathbf{Int})\})$ and $@\{\mathrm{ch}(\mathbf{Int}, t_2), \mathrm{ch}(t_2), \mathrm{ch}(\mathbf{Int})\} = [\![[\tau_2]]\!]$. Therefore e is of the form `map` $(\lambda x.e_\lambda, [v_1, ..., v_n])$ and from the observation on P' we have that $\lambda x.e_\lambda$ is applied to each element of $[v_1, ..., v_n]$.

We set $e' = [e_\lambda\{x \mapsto v_1\}, ..., e_\lambda\{x \mapsto v_n\}]$ and we know that $e \to e'$ by (B-Map). Then we have $P' \dot{\approx}_\alpha [\![e']\!]^\Gamma_o$.

Iota: For Iota we have $e = \texttt{iota}\ e_1$ and must prove that the two parts of operational correspondence hold.

Soundness: For the first part we have that if $e \to e'$ then there is a P' such that $[\![e]\!]^\Gamma_o \overset{\bullet}{\Longrightarrow} P'$ and $P' \dot{\approx}_\alpha [\![e']\!]^\Gamma_o$.

Just like `map` we have two cases. The first case follows the same argument as `map`.

For the second case we can take a step by (B-Iota) and therefore e_1 is an number. Thereby $e' = [0, 1, ..., n-1]$. Just like `map` we will go through the translation. First we create some restriction for the channels the communication will happen on.

$$\underbrace{(\nu o_1 : \mathrm{ch}([\![\mathbf{Int}]\!]))}_{\text{Input number}}.\underbrace{(\nu r : \mathrm{ch}(\mathbf{Int})).(\nu d : \mathrm{ch}(\mathbf{Int}))}_{\textit{Repeat}\text{ Process}}.$$
$$\underbrace{(\nu h : @\{\mathrm{ch}(\mathbf{Int}, \mathbf{Int}), \mathrm{ch}(\mathbf{Int}), \mathrm{ch}(\mathbf{Int})\})}_{\text{Output array}}.$$

Next we receive the value on o_1 after some administrative reductions. Then the Repeat process is started to create the array with each value being its index. Only when the Repeat process has finished will we receive on d which is an important reduction.

$$\Big([\![e_1]\!]^\Gamma_{o_1} \mid o_1(n : [\![\mathbf{Int}]\!]).\big(\mathit{Repeat}(n, r, d) \mid !r(i : \mathbf{Int}).\ \mathit{Cell}(h, i, i) \mid \bullet\, d(\_ : \mathbf{Int}).$$
$$\overline{o}\langle h\rangle \mid \overline{h.\ \texttt{len}}\langle n : \mathbf{Int}\rangle\big)\Big)$$

We then have our P' and as we output the new array we have that $P' \dot{\approx}_\alpha [\![e']\!]^\Gamma_o$.

Completeness: For the second part we have that if $[\![e]\!]^\Gamma_o \overset{\bullet}{\Longrightarrow} P'$ then there is a e' such that $e \to e'$ and $P' \dot{\approx}_\alpha [\![e']\!]^\Gamma_o$.

We know that P' has received on $\bullet\, d(\_ : \mathbf{Int})$, and as we have received on o_1 an $n : \mathbf{Int}$, e must be a value on the form n after some administrative reductions by Lemma 4.5. We then select $e' = [0, 1..., n-1]$ and by (B-Iota) we have that $e \to e'$ and $P' \dot{\approx}_\alpha [\![e']\!]^\Gamma_o$.

Size: For Size we have $e = \texttt{size}\ e_1$ and must prove that the two parts of operational correspondence hold.

Soundness: For the first part we have that if $e \to e'$ then there is a P' such that $[\![e]\!]^\Gamma_o \overset{\bullet}{\Longrightarrow} P'$ and $P' \dot{\approx}_\alpha [\![e']\!]^\Gamma_o$.

Just like `map` we have two cases. The first case follows the same argument as `map`.

For the second case we can take a step by (B-Size) and therefore e_1 is an array. Thereby $e' = n$. In the translation we receive the array handle on o_1 and by Lemma 4.5 we have that the sub-expression will output on its respective channel after som administrative reductions. We then receive the length of the array on $h \cdot$

`len` as n followed by the important reduction and then our $P' = \overline{n}\langle o \rangle$. We then have that $[\![e]\!]_o^\Gamma \overset{\bullet}{\Longrightarrow} P'$ and $P' \dot{\approx}_\alpha [\![e']\!]_o^\Gamma$.

Completeness: For the second part we have that if $[\![e]\!]_o^\Gamma \overset{\bullet}{\Longrightarrow} P'$ then there is a e' such that $e \to e'$ and $P' \dot{\approx}_\alpha [\![e']\!]_o^\Gamma$.

We have that $P' = \overline{o}\langle n \rangle$. We know that P' has received on o_1 and given we are well-typed that $o_1 : \mathrm{ch}(@\{\mathrm{ch}(\mathbf{Int}, [\![\tau]\!]), \mathrm{ch}([\![\tau]\!]), \mathrm{ch}(\mathbf{Int})\}) = [\![[\tau]]\!]$. Therefore e is on the following form: $[v_1, ..., v_n]$. We then select $e' = n$ and by (B-Size) we have that $e \to e'$ and $P' \dot{\approx}_\alpha [\![e']\!]_o^\Gamma$.

□