Resumé

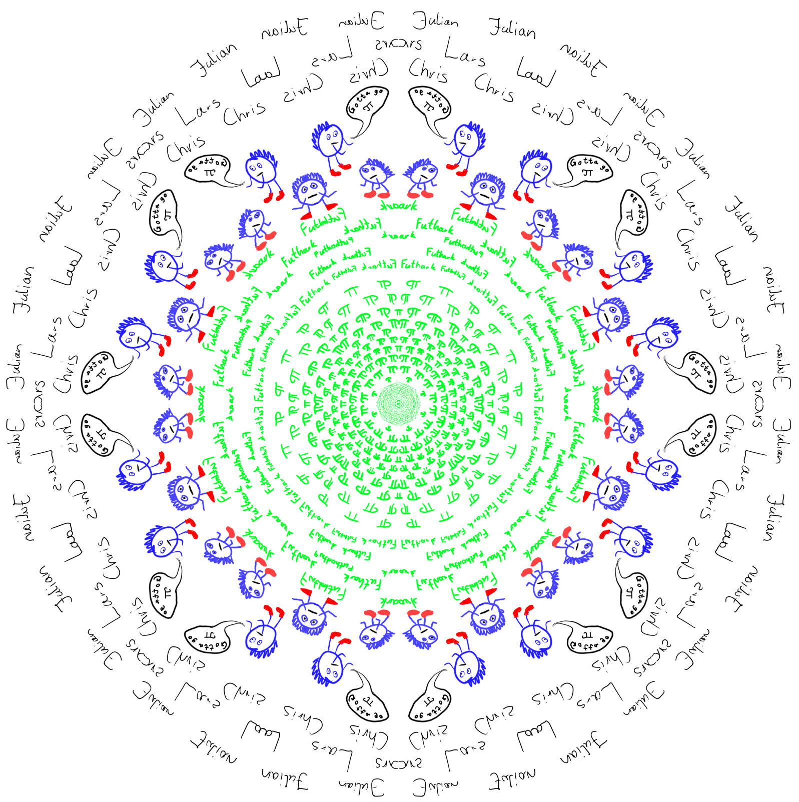
Denne rapport omhandler skabelsen af en oversættelse fra det funktionelle sprog FUTHARK til en proces kalkyle (applied π-calculus). Her tager vi udgangspunkt i det data-parallelle programmeringssproget FUTHARK, som netop prøver at gøre det nemmere for programmører at udtrykke programmer der afvikles parallelt på fysisk hardware. Vi foreslår sproget BUTF, som er en afgrænsning af FUTHARK, idet vi vælger at fokusere på arrays og en delmængde af de forskellige array operatorer FUTHARK tilbyder. Sammen med arrays har BUTF inkluderet gængse processerings-funktioner, også kaldet anden ordens array operatorer (SOACs), som dækker over en række af funktionaliteter som ofte bruges. Et gængs eksempel er at man iterer over et array og processerer element for element, men hvis iterationerne er uafhængige af hindanden, kan man bruge en map SOAC der evaluerer samtlige elementer individuelt og parallelt.

π-kalkylen i sig selv udvides med sammensatte navne og rundkastning, for at lettere kunne udtrykke en flad array struktur. Der er allerede eksisterende forskning omkring oversættelser mellem funktionelle sprog og π-kalkylen, men ikke med fokus på array datastrukturen samt SOACs. Alle konstruktioner i BUTF er blevet fremstillet i denne kalkyle, og danner dermed en fuld oversættelse. I et forsøg på at forsimple BUTF i forhold til et bevis af korrekthed, produceres der en analyse af sprog primitiver. Denne bliver brugt til se om kan aflede primitiver gennem andre primitiver uden at drastisk øge den asymptotiske kompleksitet, og derved indskrænke mængden af nødvendige primitiver.

Vi viser korrektheden af denne oversættelsen ved at opstille en ækvivalens af processer, samt en operationel korrespondance. Her introduceres distinktionen mellem administrative og vigtige reduktioner i processer, hvor vi sikrer os at reduktioner i BUTF følger vigtige reduktioner i en oversættelse.

Til sidst fremstilles der en gennemgang af de idéer der er blevet afdækket i løbet af rapporten, samt en række ideer for videre arbejde. Dette kunne bestå af kompleksitets analyse af oversættelsen, eller nærmere analyse af oversatte programmer.

Translating Concepts of the Futhark Programming Language into an Extended π-Calculus



Master's Thesis

Lars Jensen
Chris Oliver Paulsen
Julian Jørgensen Teule

Aalborg University
Computer Science Software

AALBORG UNIVERSITY
STUDENT REPORT

Computer Science and Software
Aalborg University
www.aau.dk

Title:
Translating Concepts of the Futhark Programming Language into an Extended π-Calculus

Theme:
Syntax and semantics

Project period:
Spring Semester 2023

Project group:
cs-23-sv-10-05

Participants:
Lars Jensen
Chris Oliver Paulsen
Julian Jørgensen Teule

Supervisor:
Hans Hüttel

Pages: 82

Date of completion:
June 15, 2023

Abstract:

The ever-increasing processing power of computers is evolving towards horizontal scaling. While it was normal for consumer class CPUs to be quad cores just ten years ago [1], current generations of CPUs have up to 24 cores in them [2]. This trend in computer hardware leads to the ever-growing need to properly parallelize workloads to efficiently utilize both current and also future hardware. Futhark remedies this problem by providing a comprehensive abstraction layer to the parallel hardware. It does this by providing the programmer with the data-parallel language constructs to express concurrent operations. To analyze the concurrent nature of this language, a translation into a π-calculus is proposed, due to its natural form of expressing concurrency. This report covers the design and implementation of two languages, a simple version of FUTHARK (BUTF) and an extended π-calculus (Eπ). After this, a simplification of primitives is provided to allow for a simpler translation. This translation is then shown to be correct in regard to an operational correspondence.

Preface

This report was created by the 10th semester group cs-23-sv-10-05 as a master thesis and is a continuation of their 9th semester project.
Citations appear as [1] when something is cited. Tables, figures and source code listings are numbered after the chapter number in chronological order, for instance figure 1.1 is the first figure in chapter 1.

We would like to thank the Futhark researchers at DIKU for the FUTHARK language and their time. Special thanks go to Troels Henriksen and Cosmin Oancea for their ideas, inspiration, and an interesting talk.

Aalborg University, June 15, 2023

Contents

1 Introduction **1**

2 Introduction of Languages **5**
- 2.1 The Array Language Futhark . . . 5
- 2.2 Basic Un-Typed Futhark . . . 7
  - 2.2.1 Syntax . . . 7
  - 2.2.2 Small-Step Semantics . . . 9
- 2.3 Extended π-Calculus . . . 13
  - 2.3.1 Syntax . . . 13
  - 2.3.2 Semantics . . . 15
  - 2.3.3 Weak Bisimilarity . . . 18

3 Translation of Basic Un-typed Futhark **21**
- 3.1 Correctness Criteria . . . 21
- 3.2 Translation . . . 22
  - 3.2.1 Expressions . . . 22
  - 3.2.2 Arrays . . . 25
  - 3.2.3 Second-Order Array Combinators . . . 29

4 Correctness **33**
- 4.1 Choice of Primitives . . . 33
  - 4.1.1 Recursion in Basic Un-Typed Futhark . . . 34
  - 4.1.2 Map . . . 35
  - 4.1.3 Reduce . . . 36
  - 4.1.4 Scan . . . 37
  - 4.1.5 Size . . . 38
  - 4.1.6 Iota . . . 39
  - 4.1.7 Concat . . . 41
  - 4.1.8 Selecting Primitives . . . 41
- 4.2 Administrative Reductions . . . 46
- 4.3 Weak Barbed Bisimulation . . . 48

4.4 Proving Correctness 52
4.4.1 Relaxing Operational Correspondence 52
4.4.2 Correctness of Translation 53

5 Conclusion 63
5.1 Contribution and Results 63
5.2 Basic Un-Typed Futhark and Extended π-Calculus 64
5.3 Translation 66
5.4 Future Work 68

Bibliography 71

A Basic Un-Typed Futhark Small-Step Semantics 75

B Summary of the Extended π-Calculus 79

Chapter 1

Introduction

Parallel computing is an important milestone in computing and has since the first graphics accelerators, and later the first dual-core central processing units (CPUs), been a point of interest in computer science. Nowadays, most tasks are run on multi-core machines that would benefit from utilizing all available resources. However, there are difficulties to overcome when attempting to utilize multiple cores in parallel to handle a task. The most fundamental problem is that not all tasks can be fully parallelized. Some instructions can rely on others which essentially builds a non-parallelizable chain of instructions [3]. Other problems include race conditions that need to be avoided when accessing or writing a resource multiple times.

Current hardware and their problems As graphics processing units (GPUs) get more sophisticated in terms of supported operations and higher computational unit counts, efficient parallelization becomes even more important. However, since GPUs are still physically limited in their computational power and resources, optimizations to utilize said resources become complex, especially since there are many different GPUs. Different manufacturers, architectures, bandwidths, core counts, and memory sizes are some of the parameters that need to be considered when one wants to utilize a GPU in the most efficient way.

Languages targeted at hardware OpenCL [4] and CUDA [5] are two frameworks that are commonly used to allow the programmer to write code for GPUs. While these frameworks provide the tools necessary to efficiently use GPUs, they require the programmer to often manually parallelize tasks and target specific hardware, with their unique features, to achieve the best performance possible [6].

The diversity in computer hardware justifies using an abstraction layer one can use to implement the desired functionality, whereafter a compiler can then optimize the given code for the target computer at hand. This optimization process is an ongoing endeavor since a generic compiler does not always interpret the intention of the programmer in the best possible way. [6, 7, 8]

Henriksen presents the Futhark language that focuses on the ability to express parallelizable operations naturally and therefore makes it easier for a compiler to utilize resources better [7]. Futhark is a pure functional programming language with a focus on the data-parallel array data structure. It does so by abstracting common patterns on array manipulation by providing constructs in the form of built-in functions. These include the second-order array combinators (SOACs): map, reduce, and scan, also known from other languages. These functions provide the necessary flexibility to minimize the overhead of parallelization, such that the compiler can better utilize the hardware at hand. Futhark makes use of this variable amount of parallelization by mainly targeting general purpose GPUs (GPGPUs) which are more optimized for high thread-count processing. [7]

Since Futhark is a functional language it has its roots in the λ-calculus where every computation is expressed as an application of functions. This paradigm is usually known for not allowing side effects [10, ch. 12] nor native parallelism without extensions. Futhark extends the λ-calculus, with the notion of arrays. Arrays are a construct often associated with imperative languages, where their flat structure often represents a real strip of computer memory, which makes its indexing functionality efficient on computer hardware. By indexing arrays, random memory locations can be accessed in constant time [11]. Functional languages such as Haskell or Scheme instead have lists, which are immutable and are based on nested pairs [12]. Compared to this, the arrays of Futhark allow for a data-parallel manipulation of arrays, i.e. a single operation is applied to each array element in parallel. Henriksen achieves data-parallelism, by translating Futhark programs to GPU-code through either CUDA or OpenCL. GPU hardware has limited resources for parallelization, and Futhark programs are therefore optimized by the compiler to efficiently utilize the hardware in question [7, 13]. This is because GPUs are bound by physical constraints, and it would thus be interesting to analyze these Futhark programs without the limitations of the hardware.

With this, the question arose of how to transform Futhark into a language that is hardware-independent, while still retaining the important aspects of the language.

The π-calculus To answer this question the π-calculus, which describes processes with unbounded concurrency, is considered. In this report, parallelism is meant as the act of running two processes simultaneously, while concurrency is meant as the concept of independent operations. The concept of the π-calculus is that processes can reduce concurrently. These processes consist of channels that are used to send and receive messages between processes. Here, as opposed to both λ-calculus and GPGPUs instructions, computations are expressed exclusively via communications between processes. Due to this natural representation of concurrency, the π-calculus can make the concurrent behavior of FUTHARK programs explicit and might assist in the analysis of programs.

Translation This work translates a subset of the FUTHARK to the π-calculus. To do this, we construct said subset, Basic Un-Typed Futhark (BUTF), to isolate the most interesting aspects of FUTHARK and simplify the translation. These aspects are the SOACs as well as the array data structure, while other aspects such as in-place updates and fusion rules are omitted. As the target of the translation the π-calculus is extended (Eπ) to allow for better representations of FUTHARK data structures. Eπ is based on the applied π-calculus proposed by Abadi, Blanchet, and Fournet [14].

With this translation, an analysis of its primitives is conducted which analyzes methods to reduce the set of primitives to be able to reduce the complexity of proofs. We then propose an operational correspondence, ensuring that the translation behaves correctly.

The translation and accompanied notion of correctness is inspired by related work on translation of various calculi into the π-calculus and proving the correctness of translations by Milner, Sangiorgi, and Amadio, Thomsen, and Thomsen [15, 16, 17].

Previous work This report is based on our previous work of Jensen, Paulsen, and Teule [9], which defines prototypes of the two languages (BUTF and Eπ) for a possible translation. The previous work also looked into defining interesting expressions and data structures from BUTF in Eπ and thereby indicated the feasibility of a translation. Here, we found that composite names and broadcasting provided a natural encoding of arrays and indexing [18, 19].

Overview Chapter 2 shows the two main languages used in this report, those being BUTF and Eπ. Chapter 3 provides a simple correctness relation, as well as a full translation of BUTF into Eπ. Chapter 4 demonstrates how the set of primitives in BUTF can be reduced, introduces the concept of administrative reductions as well as a new equivalence relation, and proves the correctness of the translation.

Finally chapter 5 contains a summary of the contribution made in this report, a discussion of this contribution as well as ideas for future work.

Chapter 2

Introduction of Languages

The Futhark project provides a language that eases the process of programming GPUs. The FUTHARK language is feature rich and therefore this chapter introduces a simpler version to ease translation.

This chapter also improves upon the explicitly concurrent language Eπ from our previous work. Eπ is an extended version of the applied π-calculus by Abadi, Blanchet, and Fournet [14] with broadcasting and composite names, to enable a natural encoding of arrays. [9]

2.1 The Array Language Futhark

FUTHARK is a purely functional data-parallel array language, created to simplify programming for GPUs. It employs a static, polymorphic type system which not only ensures that functions and operators are used correctly but also allows for in-place updates and size-dependent types.

FUTHARK's use of SOACs allows its compiler to optimize the code through fusion rules. These rules describe some SOACs patterns, for example, a sequential relation. By applying the rules matching the corresponding patterns, the program flow is rewritten to reduce the number of memory accesses while utilizing as much parallelism as optimally possible. An example is the fusion of the nested **map**s in equation (2.1).

$$\textbf{map}\ f\ (\textbf{map}\ g\ arr) = \textbf{map}\ (\lambda x.f\ (g\ x))\ arr \tag{2.1}$$

Here, the two functions of each **map** are combined into a single map with a function composition to eliminate the intermediate step where the outer **map** waits for the result of the inner **map**. [20]

Fusion rules are also used to introduce a chunking parameter to functions that enable the functions to be split into threads where each thread is responsible for a chunk of the array instead of processing the entire array in a single thread. [21]

Through the utilization of chunking, the chunks can be mapped to the parallel hardware which allows for a smaller work-span of a computation, since more steps are calculated simultaneously. As illustrated in figure 2.1, the span to reduce an array of eight elements ($[3,3,3,3,3,3,3,3]$) with multiplication, results in the calculation $(3\cdot 3\cdot 3\cdot 3\cdot 3\cdot 3\cdot 3\cdot 3 = 6561)$ with seven multiplication operations. The reduce function relies on the operator being associative, such that the order of operations can be manipulated and even be split. By calculating multiple sub-results at the same time, the asymptotic span to get the result is reduced from $\mathcal{O}(n)$ to $\mathcal{O}(log_2(n))$.

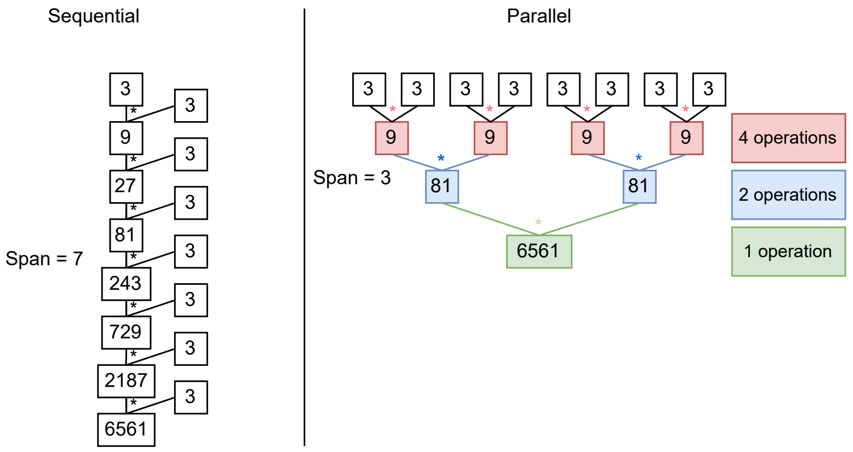


Figure 2.1: An illustration of the work-span of a problem. Inspired by [9].

$$
\begin{aligned}
\textbf{reduce}\ (*)\ 1 \quad & [3,3,3,3,3,3,3,3] \rightarrow_{chunk} \\
& [[3\cdot 3],[3\cdot 3],[3\cdot 3],[3\cdot 3]] \rightarrow \\
& [[[3\cdot 3]\cdot[3\cdot 3]],[[3\cdot 3]\cdot[3\cdot 3]]] \rightarrow \\
& [\underline{[\underline{[\underline{3}\cdot\underline{3}]}\cdot\underline{[\underline{3}\cdot\underline{3}]}]\cdot[\underline{[\underline{3}\cdot\underline{3}]}\cdot\underline{[\underline{3}\cdot\underline{3}]}]}]
\end{aligned}
$$

Figure 2.2: An example program to demonstrate parallel processing.

To illustrate conceptually how FUTHARK makes use of this, figure 2.2 shows an example program. It shows how FUTHARK can rewrite a program to an optimal concurrency to utilize the hardware's available parallelism. In case the system does not have four or more cores it would rewrite the expression to a more sequential approach, for example, $[3\cdot 3\cdot 3\cdot 3]\cdot[3\cdot 3\cdot 3\cdot 3]$ where only two cores are utilized. This reduces the overhead since in the case of the fully concurrent expression on a machine with two cores, each core would need to calculate and return $[3\cdot 3]$ twice, followed by another calculation on the results, which comes with unnecessary memory operations for intermediate variables. Therefore, the larger singular instruction set can reduce the overhead if there is not enough available parallelism. The process shown omits the more detailed process of using fusion rules and stream SOACs[1] instead of reduce to illustrate the intuition. [21, 13]

2.2 Basic Un-Typed Futhark

FUTHARK is a complex language with many features and functionalities. To abstract from less relevant aspects, the features in focus are isolated into a smaller and simpler language. This reduces the complexity of a translation while maintaining aspects such as concurrency, arrays, pure functions, and SOACs. This language (BUTF) is inspired by the prior work from [9], with some additions and changes to accomplish a better and more useful translation.

The main changes are the inclusion of tuples, the removal of the **prog** and **def** syntactic constructs, and the addition of a small-step semantics that aids in describing the bisimulation of a corresponding translation. Also, note the inclusion of tuples and pattern-matching, which provides the flexibility of binding individual values of tuples to dedicated variables while also allowing to use a single variable name for an entire tuple. This can be seen in example 2.2.1.

Example 2.2.1
The equation below shows three examples of the usage of tuples and pattern-matching.

$$\begin{aligned}
&\textbf{let } (x,y) \;=\; (1,2) \textbf{ in } (+\; x\; y) \;\rightarrow^*\; 3\\
&\textbf{let } x \;=\; (1,2) \textbf{ in } x \;\rightarrow^*\; (1,2)\\
&\textbf{let } (x,y) \;=\; (1,\ (2,\ 3)) \textbf{ in let } (a,b) \;=\; y \textbf{ in } (x\,`{+}`\,a\,`{+}`\,b) \;\rightarrow^* 6
\end{aligned}$$

2.2.1 Syntax

The syntax has received some minor modifications. Tuples and pattern-matching are now a standard structure, to make it easier to represent FUTHARK programs that use tuples. Infix functions are now denoted with ‘e‘, thus easily allowing any expression to be used as an infix function. This can be used when providing an infix function as input to another function.

[1]For a more detailed explanation of the process please refer to “Futhark: purely functional GPU-programming with nested parallelism and in-place array updates” [21] and “Design and GPGPU performance of Futhark’s redomap construct” [13].

$e ::=$	**Expression**
$\mid c$	Constant
$\mid x$	Variable
$\mid (e_0, \ldots, e_n)$	Tuple
$\mid [\vec{e}]$	Array
$\mid e_1[e_2]$	Indexing
$\mid e_1 \text{ ‘}e_2\text{‘ } e_3$	Infix operation
$\mid \lambda p.e$	Abstraction
$\mid e_1\ e_2$	Application
$\mid \textbf{let } p = e_1 \textbf{ in } e_2$	Name binding
$\mid \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3$	Control Structures
$\mid \textbf{loop } p = e_1 \textbf{ for } x < e_2 \textbf{ do } e_3$	
$p ::=$	**Pattern**
$\mid x$	Variable
$\mid (p_1, \ldots, p_n)$	Tuple pattern
$c ::= n \mid cf$	**Constants**
$cf ::=$	**Built-in Functions**
$\mid \odot$	
$\mid \triangleright$	
$\mid$ **size**	
$\mid$ **concat**	
$\mid$ **iota**	
$\mid$ **map**	SOACs
$\mid$ **reduce**	
$\mid$ **scan**	

Figure 2.3: The syntax for BUTF. This is an iteration upon the syntax presented in "Constructs of the Futhark Programming Language Described in a pi-Calculus" [9].

Figure 2.3 shows the syntax of BUTF. Here, the expression (e) is the main structure. The numbers (n), and cf denote the different built-ins and constants for the language. Since we only have numbers as primitive values, 0 represents logical false, whereas everything else is interpreted as logical true. The constant function $\odot$ denotes standard binary mathematical functions as prefix operators. These functions include the standard arithmetic operator such as $-$, $+$, $\cdot$, $/$, and $\%$, with $/$ denoting standard integer division rounding towards zero. Furthermore, it also includes four logical operators, $\wedge$, $\vee$, $\neq$, and $=$. Likewise, the unary operators $\triangleright$ include $\neg$ and -. Note that the logical operators interpret 0 as false, and anything else as true.

The arithmetic expression $2+3\cdot 4$, is written in BUTF as seen in equation (2.2).

$$2\,`{+}`\,(3\,`{\cdot}`\,4) \tag{2.2}$$

The ‘e‘ notation allows using an expression as an infix operator, such that equation (2.2) is equivalent to equation (2.3). Note that because ‘e‘ and application are left-associative, parentheses are added to ensure that $3\cdot 4$ is calculated first.

$$+\ 2\ (\cdot\ 3\ 4) \tag{2.3}$$

The meta-variable x is a placeholder for an expression that is bound by some outside binding. Such bindings happen in **let**, **loop**, and **abstraction**. When binding, patterns (p) are used instead of variables since it allows for name binding of multiple variables from a tuple.

To assist in the formulation of a small-step semantics, the syntax of BUTF has a special run-time version of loop. The syntax in equation (2.4) allows for stating the current iteration of the loop. We also introduce a run-time syntax for applied built-in functions, such as $\mathit{size}\{[1,2,3]\}$.

$$\begin{aligned} e ::= \quad & \dots \\ & \mid [x = n]\ \textbf{loop}\ p = e_1\ \textbf{for}\ x < e_2\ \textbf{do}\ e_3 \\ & \mid cf\{e_1,\dots,e_n\} \end{aligned} \tag{2.4}$$

2.2.2 Small-Step Semantics

This section describes the operational semantics for BUTF through a small-step semantics. The entire small-step semantics can be seen in appendix A for reference.

The meta-variable v is used to denote all values, with the set of all values being denoted as $\mathcal{V}$. This contains all constants (c), abstractions, as well as arrays and tuples only containing values.

Arithmetic expressions are denoted with an underline ($\underline{n+1}$) to show that the expression is not to be regarded as syntax, but rather as a mathematical expression. This semantics makes use of a $\rightarrow$ to denote a reduction and $\rightharpoondown$ to denote a syntactic rewrite.

$$\frac{e \to e'}{[e_0, \ldots, e, \ldots e_n] \to [e_0, \ldots, e', \ldots, e_n]}\ \text{[E-ARR]} \qquad \frac{e_1 \to e_1'}{e_1[e_2] \to e_1'[e_2]}\ \text{[E-INDEX-1]}$$

$$\frac{e_2 \to e_2'}{e_1[e_2] \to e_1[e_2']}\ \text{[E-INDEX-2]} \qquad \frac{0 \leq m < n \qquad n = |[\vec{v}\,]|}{[\vec{v}\,][m] \to v_m}\ \text{[E-INDEX]}$$

$$\frac{e \to e'}{(e_0 \ldots, e, \ldots, e_n) \to (e_0, \ldots, e', \ldots, e_n)}\ \text{[E-TUP]}$$

Figure 2.4: The rules for arrays and tuples in BUTF.

Figure 2.4 shows the semantics of arrays, indexing, and tuples. The rules E-ARR and E-TUP describe how expressions inside arrays and tuples can be evaluated independently. Indexing works similarly to beta reduction, requiring that the indexed array and the index are values. However, the individual expressions (e_1 and e_2) can be evaluated independently from each other.

$$\frac{}{(\lambda p.e)\ v \to e\{p := v\}}\ \text{[E-BETA]}$$

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2}\ \text{[E-APP-1]} \qquad \frac{e_2 \to e_2'}{e_1\ e_2 \to e_1\ e_2'}\ \text{[E-APP-2]}$$

Figure 2.5: The application rules for BUTF, describing β-reduction and reductions inside application.

Figure 2.5 shows the basic rules known from the λ-calculus, including β-reduction and reduction inside application. Notice how E-BETA requires that the argument is evaluated to a value (v), thus expressing the call-by-value nature of BUTF. Application rule is left-associative to keep it the same as FUTHARK.

The notation $e\{x := v\}$ is used to denote the substitution of x with v, in the expression (e). For patterns $e\{p := v\}$ denotes a substitution of all variables in the pattern p to their respective values in v.

$$\frac{}{\textbf{let } p = v \textbf{ in } e \rightarrow e\{p := v\}}\ [\text{E-L}\textsc{et}]$$

$$\frac{e_1 \rightarrow e_1'}{\textbf{let } p = e_1 \textbf{ in } e_2 \rightarrow \textbf{let } p = e_1' \textbf{ in } e_2}\ [\text{E-L}\textsc{et}\text{-1}]$$

Figure 2.6: The semantic rules for let bindings in BUTF.

Figure 2.6 describes an alternative to the abstraction syntax for name binding as a convenience to the programmer. Notice how E-LET is similar to E-BETA from figure 2.5, and that the two following expressions are equivalent.

$$\textbf{let } x = v \textbf{ in } e \quad \Leftrightarrow \quad (\lambda x.e)\ v \tag{2.5}$$

$$\frac{}{\textbf{loop } p = e_1 \textbf{ for } x < e_2 \textbf{ do } e_3 \rightarrow [x = 0]\ \textbf{loop } p = e_1 \textbf{ for } x < e_2 \textbf{ do } e_3}\ [\text{E-L}\textsc{oop}\text{-I}\textsc{nit}]$$

$$\frac{m < n}{\begin{array}{l}[x = m]\ \textbf{loop } p = v \textbf{ for } x < n \textbf{ do } e_3 \rightarrow \\ [x = \underline{m+1}]\ \textbf{loop } p = e_3\{p := v, x := m\} \textbf{ for } x < n \textbf{ do } e_3\end{array}}\ [\text{E-L}\textsc{oop}\text{-I}\textsc{ter}]$$

$$\frac{m \geq n}{[x = m]\ \textbf{loop } p = v \textbf{ for } x < n \textbf{ do } e_3 \rightarrow v}\ [\text{E-L}\textsc{oop}\text{-F}]$$

Figure 2.7: The rules for the **loop** construct in BUTF. The remaining rules can be seen in appendix A.

The semantics for the **loop** construct is described in figure 2.7. The E-LOOP-INIT rule adds the initial binding for x which is 0 with a run-time syntax notation in the form of $[x = 0]$. This is then used by the E-LOOP-ITER rule that handles the iterating process where the result of e_3 with the correct substitutions is added to p and the x variable is incremented.

$$\frac{e_1 \rightharpoondown e_2 \qquad e_2 \rightarrow e_2'}{e_1 \rightarrow e_2'}\ [\text{E-R}\textsc{ewrite}]$$

$$\frac{}{e_1\,‘e_2‘\,e_3 \rightharpoondown e_2\ e_1\ e_3}\ [\text{E-B}\textsc{inop}] \qquad \frac{}{cf \rightharpoondown \underline{\mathsf{expand}(cf)}}\ [\text{E-E}\textsc{xpand}]$$

Figure 2.8: The rules for transforming the built-in functions in BUTF.

Figure 2.8 shows the semantics for the rewrites. Among these is using a function with two parameters as an infix binary operator. The usage of ‘e‘ is always-left associative, which can be seen in the example below. This is important for operators which are not associative. E-REWRITE enables rewriting of an expression in order to take a reduction. An expression e can therefore take any reductions which its rewrites can take. E-EXPAND shows how built-in functions are transformed into a wrapper function for the run-time built-in syntax $(cf\{\})$. The wrapper functions are used since they support currying.

$$\begin{aligned}
\odot\{v_1, v_2\} &\rightharpoondown \underline{v_1 \odot v_2} \\
\rhd\ \{v\} &\rightharpoondown \underline{\rhd\ v} \\
size\{v\} &\rightharpoondown \underline{|[\vec{v}]|} \\
concat\{v_1, v_2\} &\rightharpoondown [\vec{v_1}, \vec{v_2}] \\
iota\{v\} &\rightharpoondown [0, \ldots, \underline{v-1}] \\
map\{f, \vec{v}\} &\rightharpoondown [\overrightarrow{f\ v}] \\
reduce\{f, z, \vec{v}\} &\rightharpoondown z\,‘f‘\,v_0\,‘f‘\cdots‘f‘\,v_n \\
scan\{f, z, \vec{v}\} &\rightharpoondown [a_0, \ldots, a_n] \\
&\qquad \text{where } a_i \rightharpoondown z\,‘f‘\,v_0\,‘f‘\cdots‘f‘\,v_i
\end{aligned} \tag{2.6}$$

Equation (2.6) rewrites the run-time built-ins into their respective results. This does not have to adhere to the syntax rules, as it allows for the use of mathematical expressions, which are denoted with underline. Both binary and unary operators are rewritten to their mathematical equivalents.

Size extracts the length ($|\vec{v}|$) of arrays. **Concat** gets rewritten into a function that takes two arrays and returns one single array consisting of all elements from both arrays. **Iota** is a function to construct variable length arrays by taking an arity parameter. The resulting array consists of values that are equal to their indices. **Map** takes a function and an array and returns an array, consisting of the function applied to all individual elements. **Reduce** and **scan** transform into similar functions, both requiring a two-arity function (f) with the difference being that **scan** returns an array of intermediate results and **reduce** returns a singular value. Here, f is assumed to be an associative operator with a neutral element (z) which is required to rewrite both **scan** and **reduce**. It is assumed that z is always the neutral element of g, and including it in **reduce**{} is only used when the input array is empty. Note that the **scan** of BUTF is an *inclusive* scan, meaning that element a_i includes the value of v_i.

If an expression is unable to reduce or rewrite but is not a value it is classified as an error. For example, *size*{7} would be classified as an error, as one cannot take the array size of an integer.

2.3 Extended π-Calculus

The language used as the target for the translation is the Extended π-calculus (Eπ), presented by Jensen, Paulsen, and Teule [9] and is based on the applied π-calculus presented by Abadi, Blanchet, and Fournet [14]. This section introduces the syntax and semantics of Eπ.

2.3.1 Syntax

Eπ consists of three different language constructs: processes ($\mathcal{P}_s$), terms ($\mathcal{M}_s$), and extended processes ($\mathcal{A}_s$). Here processes communicate over channels, which are identified via names or composite names (b).

$P, Q, S, \ldots ::=$		**Process**
$\mid$	$\mathbf{0}$	Null process
$\mid$	$P \mid Q$	Parallel composition
$\mid$	$!P$	Replicated process
$\mid$	$\nu a.P$	Restriction on name
$\mid$	$b(\vec{x}).P$	Receive action
$\mid$	$\bar{b}\langle N\rangle.P$	Send action
$\mid$	$\bar{b}{:}\langle N\rangle.P$	Broadcast action
$\mid$	$I(\vec{x})$	Process identifier
$\mid$	$[M \bowtie N]\ P, Q$	Conditional

Figure 2.9: The syntax for processes.

Figure 2.9 shows the syntax for processes in Eπ. First, a process may be null ($\mathbf{0}$), which is used to denote a process that does nothing. Parallel compositions ($P \mid Q$) are used to define two processes running in parallel and are a necessity for defining a process where two sub-processes can communicate. Replication ($!P$) is used to define a process that contains an unbounded amount of the process in parallel. Declaration of new names ($\nu a.P$) are used to restrict new names to the scope of P, thus making them unknown to any outside process, unless P sends it to them. Once a new name is restricted it becomes a bound name for the following process such that $\nu a.P$ means that $a \in bn(P)$. On the contrary, $fn(P)$ is the set of names that are in P but not bound. To transmit names the receive and send actions are used. Receive ($b(x).P$) receives a number, name, or composite name on a channel (b) and binds it to the variable (x) in the process (P). ($\overline{b}\langle N\rangle.P$) sends a term ($N$) on a channel ($b$), this can only be done if another process is ready to receive on the term the message is being sent on. This means that only sending on names or composite names is allowed. To send a message to multiple processes at the same time, a broadcast ($\overline{b}{:}\langle N\rangle.P$) is used. This sends a term (N) to all processes that are ready to receive on the channel b [19]. It is useful when a term needs to be propagated to multiple processes. It should also be noted that a broadcast action can happen even if no processes are currently ready to receive, in that case, the message is simply lost. Process identifiers ($I(\vec{x})$) are used to define processes, where the input to the identifier is then bound in the process. An example can be seen in example 2.3.1.

Example 2.3.1
If $Add(x,y,r) = \overline{r}\langle x + y\rangle$, then $Add(x,y,r)$ is an identifier for a process that takes two variables and a name as input and then returns the sum of the numbers on the channel r.

Lastly, there is the conditional $[M \bowtie N]\ P,Q$, where the $\bowtie$ denotes a binary logical operator. If the condition is true it evaluates to P, otherwise it evaluates to Q.

$L,M,N,\ldots ::=$		**Term**
$\mid n$		Number
$\mid b$		Name or composite name
$\mid x$		Variable
$\mid M \odot N$		Arithmetic operation
$b ::=$		
$\mid a$		Name
$\mid a \cdot M_1 \cdot \ \ldots\ \cdot M_n$	$(n > 0)$	Composite name

Figure 2.10: The syntax for terms.

The syntax for terms can be seen in figure 2.10. A term may be a number (n), which in this language is constrained to the set of integers ($\mathbb{Z}$). It can also be a name for a channel (b), which is used to send or receive messages. Variables (x) are only used as placeholders and can be bound to either a number, a name, or a composite name. Operations ($M \odot N$) are a way to use binary operators on terms. The only binary operators allowed are the standard binary operators for integers as defined in section 2.2.1. Finally, a term may also be a composite name, which is a series of terms that forms a name for a channel. The notion of composite names is based on the work of Carbone and Maffeis [18].

$A, B, C, \ldots ::=$	**Extended process**
$\mid P$	Process
$\mid A \mid B$	Parallel composition
$\mid \nu a.A$	Restriction on name
$\mid \nu x.A$	Restriction on variable
$\mid \{^{M}/_{x}\}$	Active substitution

Figure 2.11: The syntax of extended processes.

The syntax for extended processes is similar to the process syntax and can be seen in figure 2.11. An extended process may be a normal process (P), a parallel composition of extended processes ($A \mid B$), a declaration of a new name ($\nu a.A$), a variable ($\nu x.A$), or an active substitution ($\{^{M}/_{x}\}$). All of them except active substitution behave similarly to their process counterpart, with new variable declaration behaving similarly to a name declaration. Active substitution is used to replace all instances of a variable (x) with a term (M) in all processes. Extended processes function as an environment around some contained processes, in which variables (x) are mapped to terms through restriction and substitutions.

2.3.2 Semantics

The semantics of Eπ works through the notion of structural congruence ($\equiv$) and reduction ($\rightarrow$). Structural congruence allows a rewriting of expressions, to allow for simpler reduction rules. For a process to reduce, a combination of some structural congruence and reduction rules is used. The structural congruence rules for ButF are shown in figure 2.12. [14, 22]

Rename	$A \equiv A'$ by α-conversion
Par-$\mathbf{0}$	$A \mid \mathbf{0} \equiv A$
Par-A	$A \mid (B \mid C) \equiv (A \mid B) \mid C$
Par-C	$A \mid B \equiv B \mid A$
Repl	$!P \equiv P \mid !P$
New-$\mathbf{0}$	$\nu n.\mathbf{0} \equiv \mathbf{0}$
New-C	$\nu u.\nu v.A \equiv \nu v.\nu u.A$
New-Par	$A \mid \nu u.B \equiv \nu u.(A \mid B)$ when $u \notin fv(A) \cup fn(A)$
Alias	$\nu x.\{^M/_x\} \equiv \mathbf{0}$
Subst	$\{^M/_x\} \mid A \equiv \{^M/_x\} \mid A\{^M/_x\}$
Rewrite	$\{^M/_x\} \equiv \{^N/_x\}$ when $M = N$

Figure 2.12: The structural congruence rules for Eπ. Notice the usage of equality on terms, which applies to both names and numbers. [14, 23]

The Rename rule uses α-conversion which may do a change of bound names or variables [24]. This is since substituting the free variables or names would potentially result in a change of behavior due to other processes still containing the prior name. Bound names can, however, be substituted freely, as they only exist within the scope of the process, and thus substituting these does not affect any other processes. For example, for $P \mid Q$ where $P = \overline{a}\langle b\rangle.b(c)$ and $Q = a(y).\overline{y}\langle z\rangle$, substituting a in P, ie. $P\{^c/_a\}$, would mean that Q can not receive P's message anymore.

The Par- rules define the parallel composition operator as associative and commutative and that the $\mathbf{0}$ process can be omitted. The Repl rule expands a replication to an unbounded number of parallel processes.

The New- rules describe the behavior of restriction (νa). These show that a restriction on $\mathbf{0}$ is inconsequential, that restriction is commutative, and that a restriction's scope can be rewritten to either extend or shrink it.

Alias describes how a substitution on a bound name (x) that is not used has the same effect as $\mathbf{0}$. Subst applies an active substitution to any parallel process. Active substitutions and accompanying rules can be used to introduce arbitrary substitutions, which is useful when using the reduction rules. Lastly, Rewrite describes how changing the substituting term with a different but equal term is allowed. An example of this would be to substitute $2+3$ with $4+1$ or 5.

$$\textsc{Comm} \qquad \bar{b}\langle x\rangle.P \mid b(x).Q \xrightarrow{\triangle} P \mid Q$$

$$\textsc{Broad} \qquad \bar{b}{:}\langle x\rangle.Q \mid b(x).P_1 \mid \cdots \mid b(x).P_n \xrightarrow{:b} Q \mid P_1 \mid \cdots \mid P_n$$

$$\textsc{Par} \quad \frac{A \xrightarrow{\triangle} A'}{A \mid B \xrightarrow{\triangle} A' \mid B} \qquad \text{B-}\textsc{Par} \quad \frac{A \xrightarrow{:b} A' \qquad B \updownarrow_b}{A \mid B \xrightarrow{:b} A' \mid B}$$

$$\textsc{Res} \quad \frac{A \xrightarrow{q} A' \qquad q \neq :u}{\nu u.A \xrightarrow{q} \nu u.A'} \qquad \text{B-}\textsc{Res} \quad \frac{A \xrightarrow{:b} A'}{\nu b.A \xrightarrow{\triangle} \nu b.A'}$$

$$\textsc{Struct} \quad \frac{A \xrightarrow{q} A'}{B \xrightarrow{q} B'} \quad \text{if } A \equiv B \text{ and } A' \equiv B'$$

$$\textsc{Then} \qquad [M \bowtie N]\, P,Q \xrightarrow{\triangle} P \quad \text{if } M \bowtie N$$

$$\textsc{Else} \qquad [M \bowtie N]\, P,Q \xrightarrow{\triangle} Q \quad \text{if } M \not\bowtie N$$

Figure 2.13: The extended reduction rules of extended processes, with new rules for broadcasting and normal communication over composite names. Here, q is either $\triangle$ or some $:b$.

To be able to evaluate Eπ the rules in figure 2.13 are used. Here, the $\xrightarrow{:b}$ is used to denote a broadcast reduction on channel b, $\xrightarrow{\triangle}$ is used to denote a non-broadcast reduction and $\rightarrow$ when either reduction can be used. The Comm and Broad rules describe the sending and receiving actions happening in parallel processes where the first process sends a variable and the other ones are receiving. Notice that the variable name which is sent and received must be the same, which can be ensured by structural congruence.

As mentioned, the broadcasting can reduce even if $n = 0$ meaning there are no listeners. To be able to have concurrency in the calculus the Par rule shows that a process can take a reduction independent from other parallel processes. The B-Par rule ensures that a broadcast always affects all possible processes. Restrictions, described by the Res rule, do not guard processes but rather describe the scope of a name. Likewise, the B-Res acts as a guard against broadcasts and guarantees that a broadcast reduction is only labeled as such within the scope of its name. The use of structural congruence is enabled by the Struct rule. Lastly, the Then and Else rules show how the conditionals evaluate to the respective processes.

2.3.3 Weak Bisimilarity

To describe and compare translated programs, a notion of weak bisimilarity is used. The inspiration for this relation was provided by Milner [22]. This defines equivalence using program behavior instead of program structure, making it less restrictive compared to structural congruence. In the translation, this is useful, in showing that a translation behaves correctly, without requiring it to have a certain structure.

The definition of weak bisimilarity can be seen in definition 2.3.1.

Definition 2.3.1 (Weak Bisimulation) Let R be a symmetric binary relation between processes. Then R is a *weak bisimulation* if for all pairs $(P,Q) \in R$, it holds that if $P \to P'$ then there exists a Q' such that $Q \to^* Q'$ and $(P',Q') \in R$.

Also, two processes (P and Q) are *weakly bisimilar*, ie. $P \approx Q$, if there exists a *weak bisimulation* R, such that $(P,Q) \in R$.

Example 2.3.2 shows weak bisimulation and the usage of witnesses, here R^s is used to denote a symmetrical binary relation, such that $R^s = R \cup \{(y,x) \mid (x,y) \in R\}$.

Example 2.3.2
Consider the processes $P = \mathbf{0}$ and $Q = a(s).\overline{b}\langle s\rangle \mid \overline{a}\langle 5\rangle \mid \overline{a}\langle 6\rangle$. Here $P \approx Q$ with the witness $R = \{(P,\overline{b}\langle 5\rangle \mid \overline{a}\langle 6\rangle),(P,\overline{b}\langle 6\rangle \mid \overline{a}\langle 5\rangle),(P,Q)\}^s$. In this case $(P,Q) \in R$, and for any Q', P can match with zero reductions such that $(P,Q') \in R$. Vice versa, P cannot take any reductions, and R is, therefore, a weak bisimulation.

One would hardly consider P and Q in the example to behave similarly, in that Q can communicate both internally and externally, while P does nothing. By itself, weak bisimulation, as defined here, is, therefore, rather useless, in that it equates wildly different processes with each other.

To implement a relation that is more restrictive than weak bisimulation, we require that P and Q can communicate on the same channels through the notion of observability.

Definition 2.3.2 (Observability) A prefix (α) can stand for any action: when receiving on a channel (b), α becomes b and when sending or broadcasting, α becomes $\overline{b}$. A process is *observable* at a prefix α if this prefix occurs unguarded and its name is free in P. This is written as $P \downarrow_\alpha$ where $\downarrow_\alpha$ is called a *barb*.

To ensure that weak bisimilar programs behaves similarly to an outside process, a notion of *weak barbed bisimilarity* is introduced, requiring that P and Q also share barbs. Additionally, we require that P and Q are indistinguishable by any outside process, which is achieved with contexts. A context ($C \in \mathcal{C}$) is a one-hole process placed around another process (P) thus yielding a new process, ie. with $C = \overline{x}\langle 5\rangle.[]$ then $C[P] = \overline{x}\langle 5\rangle.P$.

As a starting point the barbed bisimulation described by [25] is used.

Definition 2.3.3 (Weak Barbed Bisimulation) Let R be a binary relation between processes such that R is symmetric. Then R is a *weak barbed bisimulation* if for all pairs $(P,Q) \in R$ the following holds:

1. if $P \to P'$ then there exists a Q' such that $Q \to^* Q'$ and $(P',Q') \in R$.
2. for all contexts (C), $(C[P],C[Q]) \in R$,
3. and for all prefixes (α) if $P \downarrow_\alpha$ then $Q \to^* \downarrow_\alpha$.

By requiring that P and Q are similar by any context, we require that values sent on observable channels are also equal. Example 2.3.3 shows how the iteration over contexts ensures that P and Q are indistinguishable from an outside process.

Example 2.3.3
Let $P = \overline{a}\langle 5\rangle$ and $Q = \overline{a}\langle 6\rangle$. Ignoring context, the relation $R = \{(P,Q)\}$ is a weak barbed bisimulation in that neither P nor Q can reduce and they are both only observable on $\overline{a}$. However, we can use the context $C = [] \mid a(v).[v = 5]\ \overline{b}\langle\rangle,\overline{c}\langle\rangle$ to distinguish between P and Q. Thus, having $(C[P],C[Q])$ in R would break the rules for weak barbed bisimulation, in that $C[P]$ would observe $\overline{b}$ and $C[Q]$ would observe $\overline{c}$.

Chapter 3

Translation of Basic Un-typed Futhark

Before any translation can be deemed a correct translation, it is necessary to define the correctness criteria. This chapter covers such a definition for correctness when translating from ButF into Eπ, a generic translation of all expressions in ButF.

3.1 Correctness Criteria

We consider the translation to be correct when it preserves the reduction sequence and result of the program. To do this we define operational correspondence, which preserves the reduction sequence and thus ensures translation correctness.

The translating approach in this report is inspired by Milner [15] where the translation is given a channel name, on which the translated process returns its result. The notation for translation, therefore, becomes $[\![e]\!]_o$, where o is the return channel of the expression. [15]

The definition of operational correspondence can be seen in definition 3.1.1 and is inspired by Sangiorgi [16].

Definition 3.1.1 (Operational Correspondence) A translation $[\![]\!]_o$ upholds *operational correspondence*, if it is both sound and complete.

1. (*Soundness*) If $e \rightarrow e'$, then $[\![e]\!]_o \rightarrow^* [\![e']\!]_o$
2. (*Completeness*) If for any P such that $[\![e]\!]_o \rightarrow P$ then there exists an e' such that $e \rightarrow e'$ and $P \rightarrow^* [\![e']\!]_o$.

This operational correspondence requires that a translation is both complete and sound. Here, soundness is achieved by guaranteeing that all reductions that happen in a ButF program can be matched by a sequence of reductions in the corresponding Eπ process. Since the translation might require more reductions to represent ButF behavior, a $\rightarrow^*$ arrow is used. Completeness ensures that all possible reductions from

$[\![e]\!]_o$ eventually lead to $[\![e']\!]_o$. This guarantees that a translation, cannot reduce in a manner that would not follow the original expression.

3.2 Translation

This section proposes a general translation of BUTF expressions into Eπ. BUTF and Eπ are conceptually different, in that BUTF expressions can evaluate to a value, which can be combined with other values, as seen below.

$$(\lambda x. + \ x \ 2) \ \underbrace{(\cdot \ 7 \ 8)}_{\text{is } 56} \tag{3.1}$$

Here, values and expressions are combined through operators and applications. Each expression can be said to produce a result (or continue forever), which can be used as a stand-in for the expression. In equation (3.1), $(\cdot\ 7\ 8)$ always produces 56, which can be used as a stand-in for $(\cdot\ 7\ 8)$.

In Eπ this is different. Here, values are instead communicated between processes. The contribution of a process is not the reduced process it becomes, but its communication with other processes. The rewrites in BUTF are performed by the translation, instead of being done semantically. This is mainly because it is easier for the translation to rewrite expressions, and this reduces the complexity of the translation.

The translation is categorized into three sub-sections: section 3.2.1 covers the translation of general BUTF constructs, section 3.2.2 is the translation of array-related constructs, and section 3.2.3 is the translation of SOACs.

3.2.1 Expressions

Figure 3.1 shows how the values of BUTF are expressed and sent over the return channel o in Eπ. Here, all numbers and variables are compatible with terms in Eπ, and can immediately be sent on o. The values of tuples in BUTF are first evaluated by listening on return channels o_0 to o_n and then sent as a tuple term on o. Notice that tuples listen on the handle via the composite name $h \cdot h$ to ensure that the handle h can only be used when pattern-matching. Notice also how the translation delays the return on channel o, to ensure that each expression e_i has been evaluated and added to the array, as otherwise the unfinished array would be returned.

$$
\begin{aligned}
[\![n]\!]_o &\overset{\text{def}}{=} \overline{o}\langle n\rangle && \text{Number}\\
[\![x]\!]_o &\overset{\text{def}}{=} \overline{o}\langle x\rangle && \text{Variable}\\
[\![(e_0,\ldots,e_n)]\!]_o &\overset{\text{def}}{=} \begin{array}{l} \nu o_0.\cdots.\nu o_n.\nu h.(\\ \quad \prod_{i=0}^{n}([\![e_i]\!]_{o_i}) \mid \\ \quad o_0(v_0).\cdots.o_n(v_n).(\overline{o}\langle h\rangle \mid !\overline{h \cdot h}\langle v_0,\ldots,v_n\rangle) \\ ) \end{array} && \left.\vphantom{\begin{array}{l}a\\a\\a\\a\end{array}}\right] \text{Tuple}
\end{aligned}
$$

Figure 3.1: The translation of constant values, variables, and tuples.

Figure 3.2 shows the translation of possible operations. This includes the rewriting of infix operators and currying of constant functions. The latter uses the expand function, to create an abstraction with appropriate inputs. Lastly, it translates applied unary and binary built-ins ($\odot\{x,y\}$ and $\rhd\{x\}$) into the Eπ equivalent sending on channel o.

$$
\begin{aligned}
[\![e_1`e_2`e_3]\!]_o &\overset{\text{def}}{=} [\![e_2\ e_1\ e_3]\!]_o && \text{Infix operation}\\
[\![cf]\!]_o &\overset{\text{def}}{=} [\![\underline{expand(cf)}]\!]_o && \text{Expand}\\
[\![\odot\{x,y\}]\!]_o &\overset{\text{def}}{=} \nu o_1.\nu o_2.([\![x]\!]_{o_1} \mid [\![y]\!]_{o_2} \mid o_1(x).o_2(y).\overline{o}\langle x \odot y\rangle) && \text{Built-in of binary}\\
[\![\rhd\{x\}]\!]_o &\overset{\text{def}}{=} \nu o_1.([\![x]\!]_{o_1} \mid o_1(x).\overline{o}\langle \rhd x\rangle) && \text{Built-in of unary}
\end{aligned}
$$

Figure 3.2: The translation of built-in expansion and operators.

Abstraction and application are fundamental for the functionality of BUTF and their translation can be seen in figure 3.3. The abstraction translation makes use of a new name (f) to provide a handle for the parameter. This handle requires the parameters and a return channel (ret), which are then substituted in the body of the abstraction ($[\![e]\!]_{ret}$).
On the other hand, the application translation consists of the parallel composition of the two translations together with a process that connects the outputs.

$$\begin{aligned}
[\![\lambda x.e]\!]_o &\overset{\text{def}}{=} \nu f.(!f(x,ret).[\![e]\!]_{ret} \mid \overline{o}\langle f\rangle) && \text{Abst-Val}\\
[\![\lambda(x_0,\dots,x_n).e]\!]_o &\overset{\text{def}}{=} \nu f.(!f(h,ret).h\cdot h(x_0,\dots,x_n).[\![e]\!]_{ret} \mid \overline{o}\langle f\rangle) && \text{Abst-Tup}\\
[\![e_1\ e_2]\!]_o &\overset{\text{def}}{=} \nu o_1.\nu o_2.([\![e_1]\!]_{o_1} \mid [\![e_2]\!]_{o_2} \mid o_1(f).o_2(x).\overline{f}\langle x,o\rangle) && \text{Application}
\end{aligned}$$

Figure 3.3: The translation of abstraction and application. Inspired by Amadio, Thomsen, and Thomsen [17].

ButF's translation for name bindings via **let** and its control structures (**if** and **loop**) are shown in figure 3.4. For the **let** translation, the expression $[\![e_1]\!]_{o_1}$ is evaluated first, which returns via the o_1 channel to unguard and substitute the values in $[\![e_2]\!]_{o_2}$. This allows for a translation of the pattern-matching concept.

The **if** construct is translated by evaluating the predicate $[\![e_1]\!]_{o_1}$ which is used in the similar conditional control structure in Eπ. This predicate then determines whether $[\![e_2]\!]$ or $[\![e_3]\!]$ is evaluated. Here, the output channel (o) is propagated to both bodies, eliminating unnecessary communication setup.

$$\begin{aligned}
[\![\textbf{let } x = e_1 \textbf{ in } e_2]\!]_o &\overset{\text{def}}{=} \nu o_1.([\![e_1]\!]_{o_1} \mid o_1(x).[\![e_2]\!]_o) && \text{Let-Val}\\
[\![\textbf{let } (x_0,\dots,x_n) = e_1 \textbf{ in } e_2]\!]_o &\overset{\text{def}}{=} \nu o_1.([\![e_1]\!]_{o_1} \mid o_1(h).h\cdot h(x_0,\dots,x_n).[\![e_2]\!]_o) && \text{Let-Tup}\\
[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]_o &\overset{\text{def}}{=} \nu o_1.([\![e_1]\!]_{o_1} \mid o_1(v).\,[v \neq 0]\ [\![e_2]\!]_o, [\![e_3]\!]_o) && \text{If}
\end{aligned}$$

Figure 3.4: The translation of the **let** binding and the control structure **if**.

Loop The Loop process identifier in figure 3.5 works by using two cases via the $[i < s]$ structure. The base case is that the iterator (i) has reached the end value (s) which results in the accumulator (x) being sent on the return channel (r). In the case where the Loop still has at least one iteration to run, the Loop body is evaluated sending the current iteration (i), the accumulator (x), and return channel (a') on the channel b. The new return channel (a') is then given to the next iteration of the loop. An example of Loop can be seen in example 3.2.1.

$$\begin{aligned}
&\mathsf{Loop}(b,i,s,a,r) \overset{\text{def}}{=}\\
&\quad a(x).\,[i < s]\ \nu a'.\bar{b}\langle x,i,a'\rangle.\\
&\qquad\quad \mathsf{Loop}(b,i+1,s,a',r),\\
&\qquad \overline{r}\langle x\rangle
\end{aligned}$$

Figure 3.5: The process identifier to define the **loop** construct. [9]

Example 3.2.1
Say that a programmer wants to add up the numbers 0 to 10 (not including 10). Here, they could use the Loop identifier as follows.

$$\text{Loop}(b,0,10,a,r) \mid \overline{a}\langle 0\rangle \mid !b(x,i,o).\overline{o}\langle x+i\rangle$$

Notice how with each iteration a request is sent to channel b with the current accumulator (x) and enumerator value (i).

The **loop** ButF construct has a few requirements, which must be reflected by the translation. It has to return $\llbracket e_1\rrbracket_{o_1}$ when the iterator (y) is lower than $\llbracket e_2\rrbracket_{o_2}$ or it has to run $\llbracket e_3\rrbracket_{o_3}$ with the iterator and the initial or recursive bindings. In the translation of **loop**, shown in figure 3.6, $\llbracket e_1\rrbracket_{o_1}$ and $\llbracket e_2\rrbracket_{o_2}$ are in a parallel composition, allowing them to be evaluated independently. To execute the **loop**, o_2 is used to receive the stop condition, whereafter the Loop process identifier then can be used in which e_1 can directly send the starting state via o_1. The sub-process $!b(x,y,o_3).\llbracket e_3\rrbracket_{o_3}$ calls the body of the loop, binding the names x and y.

$$\left\llbracket \begin{array}{l} \textbf{loop } x = e_1 \\ \quad \textbf{for } y \ < e_2 \textbf{ do } e_3 \end{array} \right\rrbracket_o \overset{\text{def}}{=} \left.\begin{array}{l} \nu b.\nu o_1.\nu o_2.(\llbracket e_1\rrbracket_{o_1} \mid \llbracket e_2\rrbracket_{o_2} \mid \\ \quad !b(x,y,o_3).\llbracket e_3\rrbracket_{o_3} \mid \\ \quad o_2(s).\text{Loop}(b,0,s,o_1,o) \\ ) \end{array}\right] \text{Loop-Val}$$

$$\left\llbracket \begin{array}{l} \textbf{loop } (x_0,\ldots,x_n) = e_1 \\ \quad \textbf{for } y \ < e_2 \textbf{ do } e_3 \end{array} \right\rrbracket_o \overset{\text{def}}{=} \left.\begin{array}{l} \nu b.\nu o_1.\nu o_2.(\llbracket e_1\rrbracket_{o_1} \mid \llbracket e_2\rrbracket_{o_2} \mid \\ \quad !b(h,y,o_3).h\cdot h(x_0,\ldots,x_n).\llbracket e_3\rrbracket_{o_3} \mid \\ \quad o_2(s).\text{Loop}(b,0,s,o_1,o) \\ ) \end{array}\right] \text{Loop-Tup}$$

Figure 3.6: The translation of **loop** control structure.

3.2.2 Arrays

The **array** construct is the core data structure for ButF and thus needs to be implemented in Eπ. The process identifier for array is therefore carefully constructed, so it reflects the advantages of arrays. Each element in the array structure is independent of each other due to the parallel composition of elements.

The Array in figure 3.7 requires two channels and a number at initialization (*handle*, *write*). Here, the channel *handle* provides access to the array since the array always sends its read channel and length on it. This means that any process with the handle can ask for the channel *read*, which provides the functionality of the array, and its length (*len*) as value. The composite name ($\overline{handle \cdot index}$) is used to index the array where the requester uses the handle together with the desired index to listen to the value. The *write* channel is used to add elements to the array whereas the *read* channel sends all the array's elements on the provided channel.

The parameter *len* is expected to be a number given when the array is created, and can later be retrieved via *handle*. Otherwise, it would be hard to count the number of elements received at *write* without some sort of stop signal. Given that arrays in BUTF always have a known static size at runtime, the static *len* parameter reflects this. An example of the identifier can be seen in example 3.2.2.

$$
\begin{aligned}
&\mathsf{Array}(handle, write, len) \overset{\text{def}}{=} \\
&\quad \nu read.\nu b.(\\
&\qquad !write(index, v).(\\
&\qquad\quad !b(r).\overline{r}\langle index, v\rangle \mid \\
&\qquad\quad !\overline{handle \cdot index}\langle v\rangle \\
&\qquad) \mid \\
&\qquad !read(r).\overline{b}{:}\,\langle r\rangle \mid !\overline{handle}\langle read, len\rangle \\
&\quad)
\end{aligned}
$$

Figure 3.7: The process identifier to define the array construct. [9]

Example 3.2.2
The array [2,3,5] can be expressed with the array identifier as follows:

$$\mathsf{Array}(h, i, 3) \mid \overline{i}\langle 0, 2\rangle \mid \overline{i}\langle 1, 3\rangle \mid \overline{i}\langle 2, 5\rangle \tag{3.2}$$

Here, h denotes a handle to the array which can be used in the program.

As mentioned, the *read* channel allows requesting all elements on the array to be sent on a channel r. To achieve this, r is distributed to all the elements through a broadcast on the channel b, requiring only a single reduction. However, this assumes that all wanted array elements have been sent on *write* before sending a request on *read*, which is not always the case. This could be solved by waiting for all elements before sending on b, either as a part of *Array* or when calling *Array* as a process identifier.

The Await process identifier facilitates waiting, ensuring that all parts of an array are ready before they can be used. The identifier can be seen in figure 3.8, and is a simple recursively defined process that sends on a *done* channel once it has received the specified amount of inputs on a *count* channel.

$$\mathsf{Await}(n, count, done) \overset{\text{def}}{=} [n = 0]\ \overline{done}\langle\rangle, count().\mathsf{Await}(n-1, count, done)$$

Figure 3.8: The process identifier for Await.

In figure 3.9 the Array process identifier has been modified such that outside processes are notified when all expected elements are received.

$$\begin{aligned}
&\mathsf{Array}(handle, write, len, done) \overset{\text{def}}{=}\\
&\quad \nu read.\nu b.\nu count.(\\
&\qquad \mathsf{Await}(len, count, done) \mid\\
&\qquad !write(index, v).(\\
&\qquad\quad !b(r).\overline{r}\langle index, v\rangle \mid\\
&\qquad\quad !\overline{handle \cdot index}\langle v\rangle \mid\\
&\qquad\quad \overline{count}\langle\rangle\\
&\qquad) \mid\\
&\qquad !read(r).\overline{b}{:}\langle r\rangle \mid\\
&\qquad !\overline{handle}\langle read, len\rangle\\
&\quad)
\end{aligned}$$

Figure 3.9: An altered array implementation using Await to notify processes when the array is ready.

Figure 3.10 shows how the new Array process identifier is used to translate ButF arrays. Here, the array handle is returned when the array is done (signaled on *done*).

$$\left.\begin{aligned}
[\![[e_0, \dots, e_{n-1}]]\!]_o \overset{\text{def}}{=}\ &\nu handle.\nu write.\nu done.(\mathsf{Array}(handle, write, n, done) \mid\\
&\quad \prod_{i=0}^{n} \nu o_i.([\![e_i]\!]_{o_i} \mid o_i(v_i).\overline{write}\langle i, v_i\rangle) \mid\\
&\quad done().\overline{o}\langle handle\rangle\\
&)
\end{aligned}\right]\ \text{Array}$$

$$\left.\begin{aligned}
[\![e_1[e_2]]\!]_o \overset{\text{def}}{=}\ &\nu o_1.\nu o_2.([\![e_1]\!]_{o_1} \mid [\![e_2]\!]_{o_2} \mid\\
&\quad o_1(h).o_2(i).\,[i \geq 0]\ h \cdot i(v).\overline{o}\langle v\rangle, \mathbf{0})
\end{aligned}\right]\ \text{Indexing}$$

Figure 3.10: Translation of ButF arrays and indexing to Eπ using the Array process identifier.

Size The built-in of $size\{x\}$, uses the array handle to extract the length and returns it on r. This translation can be seen in figure 3.11.

$$[\![size\{x\}]\!]_o \overset{\text{def}}{=} \nu o_1.([\![x]\!]_{o_1} \mid o_1(h).h(\_, len).\overline{o}\langle len \rangle)$$

Figure 3.11: The translation of the **size** function.

Iota The Iota process identifier, as seen in section 3.2.2, requires a lower bound (m), an upper bound (n), and a return channel (r). The lower bound is necessary since this allows splitting it recursively, thus allowing for more concurrency. It then proceeds to check for the base case where the lower and upper bounds are equal, in which case it returns the element and identical index. If the base case is not fulfilled, the recursive case splits the work up into two sub-problems.

$$\begin{aligned}\mathsf{Iota}(m,n,r) \overset{\text{def}}{=} {} & [m = n]\ \overline{r}\langle n,n \rangle, \\ & \mathsf{Iota}(m,n/2,r) \mid \mathsf{Iota}(n/2+1,n,r)\end{aligned}$$

Figure 3.12: The process identifier for Iota.

$$[\![iota\{x\}]\!]_o \overset{\text{def}}{=} \begin{aligned} & \nu o_1.([\![x]\!]_{o_1} \mid \nu h.\nu write.\nu done.o_1(n).(\\ & \quad \mathsf{Array}(h, write, n, done) \mid \\ & \quad \mathsf{Iota}(0, n, write) \mid \\ & \quad done().\overline{o}\langle h \rangle \\ &)) \end{aligned}$$

Figure 3.13: The translation of $iota\{x\}$.

The iota translation **iota** makes use of the Iota process identifier. Figure 3.13 shows the translation of the built-in. The built-in creates a new array and uses it as the output for the Iota process identifier.

Concat The translation of **concat** in figure 3.14 takes in the two handles of the two arrays on o_1 and o_2 which are then used to get their respective *read* channels and their *len* values. A new array is created to capture the elements of both arrays but with their combined lengths. The values of the first array are sent over unchanged, by supplying $read_1$ to the input of the new array ($write_3$). For the second array, its indices need to be increased by the length of the first array. A replicated process is used as a bridge that handles the receiving of elements from the second array.

$$
[\![concat\{x,y\}]\!]_o \overset{\text{def}}{=} \begin{array}{l}
\nu o_1.\nu o_2.([\![x]\!]_{o_1} \mid [\![y]\!]_{o_2} \mid o_1(h_1).o_2(h_2). \\
\quad h_1(read_1, len_1).h_2(read_2, len_2). \\
\quad \nu h_3.\nu in_3.\nu done.(done().\bar{o}\langle h_3\rangle \mid \\
\qquad \mathsf{Array}(h_3, write_3, len_1 + len_2, done) \\
\qquad \overline{read_1}\langle write_3\rangle \mid \\
\qquad \nu x.(\overline{read_2}\langle x\rangle \mid !x(i,v).\overline{write_3}\langle i + len_1, v\rangle) \\
\quad) \\
)
\end{array}
$$

Figure 3.14: The translation of the **concat** function.

3.2.3 Second-Order Array Combinators

This section presents the translation of the different SOACs.

Map Map is simple, in that each element of the input array can be handled independently. Figure 3.15 shows the process identifier for a simple map over values. It requires a channel on which the array outputs its elements (*input*), the name of the function channel that has to be mapped to the elements (f), and a return channel (*output*). It maps the function onto the elements by receiving all elements and then sending the values on the function channel. A replicated process then receives all values, processes them, and returns the updated value on the return channel. Finally, the updated value is sent on the output channel, with its respective index.

$$
\mathsf{Map}(input, f, output) \overset{\text{def}}{=} \begin{array}{l}
!input(index, value).\nu r.(\overline{f}\langle value, r\rangle \mid \\
\quad r(value').\overline{output}\langle index, value'\rangle)
\end{array}
$$

Figure 3.15: The **map** process identifier. [9]

The translation of map, can be seen in figure 3.16, where the parameters x and y are the mapping function and input array respectively.

$$
[\![\mathbf{map}\{x,y\}]\!]_o \stackrel{\text{def}}{=}
\begin{array}{l}
\nu o_1.\nu o_2.([\![x]\!]_{o_1} \mid [\![y]\!]_{o_2} \mid \\
\quad \nu h_r.\nu \mathit{write}.\nu \mathit{done}.o_1(f_m).o_2(h).h(\mathit{read},\mathit{len}).(\\
\quad\quad \mathsf{Array}(h_r,\mathit{write},\mathit{len},\mathit{done}) \mid \\
\quad\quad \nu x.(\overline{\mathit{read}}\langle x\rangle \mid \mathsf{Map}(x,f_m,\mathit{write})) \mid \\
\quad\quad \mathit{done}().\bar{o}\langle h_r\rangle \\
))
\end{array}
$$

Figure 3.16: The translation for the **map** SOAC.

Reduce Reduce can be seen in figure 3.17. When reducing arrays all elements should be combined into a single value by using an operator. This is achieved using an accumulator value, which is sequentially updated with values from the array. However, this would result in an inefficient linear span, as each element is handled sequentially.

A better approach is to divide and conquer the problem by splitting the problem into two sub-problems of similar structure but with half the problem size. This relies on the operator being associative. This approach can be applied recursively resulting in a binary tree of span $\log_2(n)$ where each sub-problem can be calculated independently.

$$
\mathsf{Reduce}(\mathit{arr},f,\mathit{ret}) \stackrel{\text{def}}{=}
\begin{array}{l}
\nu \mathit{reduce}.\nu \mathit{read}.(\\
\quad \mathit{arr}(\mathit{read},\mathit{len}).\overline{\mathit{read}}\langle \mathit{read}\rangle.\overline{\mathit{reduce}}\langle \mathit{ret},\mathit{len}+1\rangle \mid \\
\quad !\mathit{reduce}(r,n).(\\
\quad\quad [n=1]\mathit{read}(\_,v).\bar{r}\langle v\rangle, \\
\quad\quad [n>1]\nu r'.(\\
\quad\quad\quad \overline{\mathit{reduce}}\langle r',n/2\rangle \mid \overline{\mathit{reduce}}\langle r',n-n \operatorname{div} 2\rangle \mid \\
\quad\quad\quad r'(v_1).r'(v_2).\bar{f}\langle v_1,v_2,r\rangle))))
\end{array}
$$

Figure 3.17: The **reduce** process identifier. [9]

The process identifier for reduction is seen in figure 3.17. It starts by requesting the length and the elements of the array, which it then sends along an internal *reduce* channel. The reduce happens in the replicated process where this channel is used to listen to a return channel and the current length of the work set. This allows the replication to work recursively where each set of values is split, determined by its length until the length reaches 1 ($[n = 1]$). For the recursive step when $[n > 1]$, the work set is split into two sets by sending both splits on *reduce* with a unique return channel (r') which results in a binary tree of reduces.

$$
[\![\mathit{reduce}\{x,y,z\}]\!]_o \overset{\text{def}}{=} \begin{array}{l} \nu o_1.\nu o_2.\nu o_3.([\![x]\!]_{o_1} \mid [\![y]\!]_{o_2} \mid [\![z]\!]_{o_3} \mid \\ \nu op.(o_1(f_r).o_2(z).o_3(h).h(\_, len).\,[len = 0]\; \overline{o}\langle z\rangle, \\ \quad (\mathsf{Reduce}(h, op, o) \mid \\ \qquad op(x,y,ret).\nu r.\overline{f_r}\langle x,r\rangle.r(f'_r).\overline{f'_r}\langle y, ret\rangle))) \end{array}
$$

Figure 3.18: The translation for the **reduce** SOAC.

The Built-in translation can be seen in figure 3.18. Similar to the map translation, it captures the output of its parameters and parses them into the Reduce process identifier. If the array does not contain any elements, it instead just returns the neutral element.

Scan Scan functions much like reduction, but produces intermediate results as elements are combined. Here, the i'th value of the resulting array is the reduction of elements 0 to i. The problem is tackled by a divide-and-conquer approach by defining a pivot point (p) that splits this problem as seen in equations (3.3) and (3.4).

$$
\mathbf{scan}\ f\ z\ [v_0, \ldots, v_p, \ldots, v_n] \tag{3.3}
$$

$$
\begin{array}{l} \mathbf{let}\ \mathit{first} = \mathbf{scan}\ f\ z\ [v_0, \ldots, v_p] \\ \quad \mathbf{in}\ \mathit{first}\ \text{`}\mathbf{concat}\text{`}\ (\mathbf{map}\ (f\ (\mathit{first}[p]))\ (\mathbf{scan}\ f\ [v_{p+1}, \ldots, v_n])) \end{array} \tag{3.4}
$$

With this in mind, a recursive scan process over elements with indices s to (and including) t can be defined as seen in figure 3.19.

$$
\begin{array}{l} \mathsf{Scan}(arr, f, s, t, r) \overset{\text{def}}{=} \nu r_1.\nu r_2.([s < t] \\ \quad \mathsf{Scan}(arr, f, s, s + (t - s)\ \mathsf{div}\ 2, r_1) \mid \\ \quad \mathsf{Scan}(arr, f, s + (t - s)\ \mathsf{div}\ 2 + 1, t, r_2) \mid \\ \quad !r_1(i, v).(\overline{r}\langle i, v\rangle \mid \\ \qquad [i = s + (t - s)\ \mathsf{div}\ 2]\ !r_2(i', v').\overline{f}\langle v, v', r'\rangle.r'(v'').\overline{r}\langle i', v''\rangle, \mathbf{0} \\ \quad), \\ \quad arr \cdot s(v).\overline{r}\langle s, v\rangle) \end{array}
$$

Figure 3.19: The **scan** process identifier.

Figure 3.20 shows the translation of the built-in. The function channel *op* is used to represent the operator for reduction. Given that such functions, of arity 2, are represented through currying, the last line of figure 3.20 uncurries f into a single function.

$$
[\![scan\{x, y, z\}]\!]_o \stackrel{\text{def}}{=} \begin{array}{l}
\nu o_1.\nu o_2.\nu o_3.([\![x]\!]_{o_1} \mid [\![y]\!]_{o_2} \mid [\![z]\!]_{o_3} \mid \\
\nu\, op.\nu\, write.\nu res.\nu\, done.(\\
o_1(f_s).o_2(z).o_3(h).h(\_, len).(\\
\qquad \mathsf{Scan}(h, op, 0, len - 1, write) \mid \\
\qquad \mathsf{Array}(res, write, len, done) \mid \\
\qquad done().ro(res) \mid \\
\qquad !op(v_1, v_2, r_f).\nu r'_f.\overline{f_s}\langle v_1, r'_f\rangle.r'_f(f'_s).\overline{f'_s}\langle v_2, r_s\rangle)))
\end{array}
$$

Figure 3.20: The translation of the **scan** built-in.

Chapter 4

Correctness

This chapter shows the correctness of BUTF through a proof of operational correspondence. Currently, each SOAC is translated separately as a primitive, requiring proof to handle each of these cases. Instead, the number of primitives can be reduced, by implementing some BUTF constructs in BUTF itself. Therefore, the possible subsets of primitives that are necessary are explored in this chapter. Alongside this, an annotation of reduction is proposed, a new bisimilarity based on these annotations, as well as a proof of correctness.

4.1 Choice of Primitives

In this section, the set of primitives is reduced to find a smaller set of necessary primitives, which the remaining language constructs can then be constructed from. This reduces the complexity and amount of proofs needed to determine a correct translation.

Since the goal of this analysis is not to find the smallest subset, but merely a subset that is easier to prove, this section does not attempt to prove that the subset is the smallest possible. Instead, the focus is on finding concrete dependencies between primitives.

These dependencies are then described together with the asymptotic complexities of their work ($\mathcal{W}$) and their span ($\mathcal{S}$). The list of primitive candidates consists of **map**, **reduce**, **scan**, **size**, **iota**, and **concat**. If a candidate is chosen to be a primitive, it uses the implementation presented in chapter 3, and otherwise it is derived from some other constructs. The rest of the translation except loop is as presented in chapter 3. When describing complexity for functions that depend on other primitive candidates, the notation $\mathcal{W}_f(n)$ is used to denote the worst-case work complexity of f, given an array of size n. $\mathcal{S}_f(n)$ is used similarly for span. With this notation, each BUTF reduction is defined to cost 1 work, while rewrites are free. When defining work and span, constants are omitted for simplicity.

4.1.1 Recursion in Basic Un-Typed Futhark

When deriving BUTF constructs from primitives, recursion is a useful tool. Though, explicit recursion is not supported by the version of BUTF described in chapter 2, it can be expressed through the call-by-value *fixed-point combinator*, which can be seen in listing 4.1.1 [26]. *Fix* is a generalization of the diverging process $(\lambda x.x\ x)\ (\lambda x.x\ x)$, such that the function f decides whether to expand *fix* again or not. Note how $x\ x$ has been replaced by $(\lambda y.x\ x\ y)$ as otherwise the application of *fix* would always diverge due to BUTF's call-by-value semantics [26]. Example 4.1.1 shows how the *fix* function can be used to define a factorial function.

```
let fix = \f.(\x.f (\y.x x y)) (\x.f (\y.x x y))
```

Listing 4.1.1: The fixpoint operator in BUTF.

Example 4.1.1
Recursion can then be expressed using *fix* as seen in the creation of the factorial function in listing 4.1.2.

```
let fact = fix (\f. \n. if (= n 0) then 1 else (* n (f (- n 1)))
) in fact 10
```

Listing 4.1.2: Factorial function implemented recursively using the *fix* function.

This can be used to implement **loop** by using an iterator that is propagated through the recursion and an if statement that is used to define when the recursion ends.

$$\textbf{loop}\ p = e_1\ \textbf{for}\ x < e_2\ \textbf{in}\ e_3 \overset{\text{def}}{=} \texttt{loop}\ e_2\ (\texttt{\textbackslash}p.\ \texttt{\textbackslash}x.\ e_3)\ e_1\ \texttt{0} \tag{4.1}$$

```
let loop = \n. \body. (fix (\f.
    \p. \x. if (>= x n) then p else
              (f (body p x) (+ x 1))
)) in ...
```

Listing 4.1.3: The **loop** construct implemented via recursion.

Listing 4.1.3 and equation (4.1) shows this implementation where **loop** is defined via the fix keyword where f is the handle for the recursive call, p is the initial parameter or the result of the last evaluation of the body, x is the iterator, and n

is the upper bound. By comparing the iterator (x) and upper limit (n), either p or the result of a recursive call is returned. For the recursive step, the incremented x and newly evaluated body is propagated. This means that **loop** is now replaced by recursion to ease further analysis.

4.1.2 Map

Map is one of the potential primitives that could allow for concurrency and thus its span should be as small as possible. The easiest way to achieve this is by implementing **map** as a primitive, but we also found two other ways to implement **map**.

A simple way of implementing **map** is using **scan**, since this is also a concurrent operator.

```
let map = (\f. scan ( \x. \y. f y) 0)
    in ...
```

Listing 4.1.4: Map implemented using **scan**.

This implementation can be seen in listing 4.1.4, where it uses a let statement to define **map**. This implementation then relies on a **scan** and a function that takes two inputs and discards the first one. After the first input is discarded all that is left is the function $\lambda y.fy$ which corresponds to a **map** function. Thus the end result is the same as mapping f. A downside of this method is that this only works as long as the **scan** implementation combines elements left-associatively. This is due to the fact that the operator function does not fulfill the associative property. Since this **map** is based on a **scan**, the work and span depend on the **scan** implementation. So the work is $\mathcal{O}(\mathcal{W}_{scan}(n, \mathcal{W}_f))$ and the span is $\mathcal{O}(\mathcal{S}_{scan}(n, \mathcal{W}_f))$.

Another way to implement **map** would be with **concat**, **size**, and recursion.

```
let map = (\f. \arr (
    fix (\maph. \n. \m.
        if (n ´=´ m) then [f arr[n]]
        else
            (maph n (m ´/´ 2)) ´concat´ (maph ((m ´/´ 2)´+´ 1) n)
        )
    0 (size arr))
) in ...
```

Listing 4.1.5: Map implemented using **concat**, **size** and recursion.

In listing 4.1.5, the main idea is to split the mapping into a binary tree. This is done using a helper function (*maph*) which takes an array, a start, and an end index. This function then keeps splitting into a binary tree of recursive calls, until the start and end are the same for the sub-problem at hand. Then it applies the function (f) to the element of the array and returns that as a single-element array. **Concat** is then used on the results to combine them into one single array. The complexity of *maph* can be described recursively as seen in equation (4.2), where the problem size (o) is $m-n$. Here, it is assumed that the complexity is $\mathcal{W}_f$ when $o=0$.

$$\begin{aligned} \mathcal{W}_{maph}(o,\mathcal{W}_f) &= 2\cdot\mathcal{W}_{maph}(o/2,\mathcal{W}_f)+\mathcal{W}_{concat}(o) \\ \mathcal{S}_{maph}(o,\mathcal{S}_f) &= 2\cdot\mathcal{S}_{maph}(o/2,\mathcal{S}_f)+\mathcal{S}_{concat}(o) \end{aligned} \tag{4.2}$$

Note that the two subcalls to *maph* are not done concurrently, and we, therefore, add their spans. Because the complexity of **concat** is unknown, it is hard to state a non-recursive complexity of this **map** implementation. Given that **map** calls *maph* with parameters 0 and the size of array, thus the work of **map** is $\mathcal{W}_{maph}(n,\mathcal{W}_f)+\mathcal{W}_{size}$, for array size n. The span of **map** can be expressed similarly.

Finally, **map** can also be chosen as a primitive, in which case it is assumed that it has the same asymptotic complexity as FUTHARK. For **map** this is a work of $\mathcal{O}(n\cdot\mathcal{W}_f)$ and a span of $\mathcal{O}(\mathcal{S}_f)$ [27].

4.1.3 Reduce

Reduce is a powerful and highly versatile SOAC, and its concurrent nature makes it an option when it comes to reducing span. However, **reduce** can also be derived from other constructs in multiple ways. The simplest is using **scan** and **size**, where the function used in the **reduce** is instead used in the **scan** operation. Since the last element of **scan** is the same as a **reduce** would yield, **size** can then be used to index this last element and get the usual **reduce** result. This relies on **scan**, giving an asymptotic work of $\mathcal{O}(\mathcal{W}_{scan}(n,\mathcal{W}_f)+\mathcal{W}_{size}(n))$ and span of $\mathcal{O}(\mathcal{S}_{scan}(n,\mathcal{S}_f)+\mathcal{S}_{size}(n))$. This implementation of **scan** can be seen in listing 4.1.6.

```
let reduce = \f. \n. \arr. (scan f n arr)[size arr `-` 1]
    in ...
```

Listing 4.1.6: Reduce implemented using **scan** and **size**.

Another way to construct **reduce** is by using recursion together with **size**, where the array is simply iterated over and a binary operator is used to combine the results. **Size** is used to set an upper bound the iterating process. This is a more linear approach and therefore the work is $\mathcal{O}(n\cdot\mathcal{W}_f+\mathcal{W}_{size}(n))$ while the span is $\mathcal{O}(n\cdot\mathcal{S}_f+\mathcal{S}_{size}(n))$. This implementation can be seen in listing 4.1.7.

```
let reduce = \f. \n. \arr.
    loop x = n for i < size arr do
        x = f x arr[i] in ...
```

Listing 4.1.7: The **reduce** SOAC made with **loop** and **size**.

If **reduce** is chosen as a primitive, the work and span from FUTHARK can be used, which is $\mathcal{O}(n \cdot \mathcal{W}_f)$ and $\mathcal{O}(log_2(n) \cdot \mathcal{S}_f)$ [27].

4.1.4 Scan

Scan can be used to construct **map** and **reduce**. However, a primitive **scan** is rather complex, which means having it as a derived operator eases a correctness proof considerably. It can be naïvely derived from a series of reductions as seen in equation (2.6) in section 2.2.2.

```
let scan = fix (\f. \op. \z. \arr.
   -- We need pairs, which can pair up values in list, ignoring a
 ↪  potential last odd element
   let pairs = (\arr.
      map (\i. (arr[* i 2], arr[+ 1 (* i 2)]))
         (iota (size arr `/` 2))
   )
   in let s = size arr
   if (s `<` 2) then arr
   else
      in let paired = map (\(x,y). op x y) (pairs arr)
      in let recur = scan op z paired
      in map (\i.
        let carry = if (= i 0) then z else recur[/ (- i 1) 2]
        in if (= 0 (% i 2))
           then (op arr[i] carry)
           else carry
       ) (iota s)
) in ...
```

Listing 4.1.8: The implementation of *work efficient scan* in BUTF.

With **map** and **iota**, **scan** can be implemented as the *work efficient scan* algorithm [28, 29], which can be seen in listing 4.1.8. The algorithm combines the array *arr* in pairs, creating a new array of size $\lfloor n/2 \rfloor$, which it calls **scan** on recursively. This yields the array *recur*, where the i'th value is the result of reducing values 0 to $i \cdot 2$ in *arr*. *Recur* and *arr* are then combined, to create the **scan** result where the j'th element is determined according to the following two rules.

- If j is odd, the value can be found directly in *recur* at the index $\lfloor (j-1)/2 \rfloor$. This is because *recur* contains the final result for every odd element.
- If j is even, then *recur* does not contain the result for the j'th value. Instead we add the $\lfloor (j-1)/2 \rfloor$'th value of *recur* to $arr[j]$ to yield the **scan** value at j. If j is 0, there is no $j-1$, so the neutral element (z) is added instead.

The work and span complexity of **scan** can be written as seen in equation (4.3)

$$\begin{aligned} \mathcal{W}_{scan}(n, \mathcal{W}_f) = \mathcal{W}_{scan}(n/2, \mathcal{W}_f) + \mathcal{O}(\mathcal{W}_{iota}(n) + \mathcal{W}_{map}(n, \mathcal{W}_f) + \mathcal{W}_{size}(n)) \\ \mathcal{S}_{scan}(n, \mathcal{S}_f) = \mathcal{S}_{scan}(n/2, \mathcal{S}_f) + \mathcal{O}(\mathcal{S}_{iota}(n) + \mathcal{S}_{map}(n, \mathcal{S}_f) + \mathcal{S}_{size}(n)) \end{aligned} \tag{4.3}$$

Scan when chosen as a primitive can be assumed to have the same complexity as the implementation in FUTHARK which has a work of $\mathcal{O}(n \cdot \mathcal{W}_f)$ and a span of $\mathcal{O}(log_2(n) \cdot \mathcal{S}_f)$ [30].

4.1.5 Size

Size is an interesting function since it allows for dynamic bounded indexing in recursion constructs which helps construct efficient and error-free functionality. Without **size**, going through each element of an array at a time is impractical the recursion would not know when to stop. This would at some point cause indexing of a non-existing index resulting in undefined behavior.

However, **size** can be built in two other ways using the other primitive candidates. The first is a **reduce** that uses the counting function ($\lambda x.\lambda y.x$ '+' 1) and the neutral element 0, as seen in listing 4.1.9. This creates an equation that starts with 0 and adds 1 for each element in the array, resulting in the size of the array. One flaw of this approach is that the counting function is not associative and thus only works if the **reduce** applies the function left-associatively. This implementation relies on the complexity of **reduce** for both the work and the span, making the work $\mathcal{O}(\mathcal{W}_{reduce}(n,1))$ and span $\mathcal{O}(\mathcal{S}_{reduce}(n,1))$.

```
let size = \arr. reduce(\x. \y. x ´+´ 1) 0 arr
    in ...
```

Listing 4.1.9: Size implemented with **reduce**.

The second way **size** can be defined is by using **map**, **concat**, and recursion. The intuition is to create an array with the same length as the input array where each value is replaced by some non-zero value. To accommodate the lack of **size**, each element is mapped to some non-zero ($\neg$ 0) value, and a zero is concatenated to the end. Then the size can be found by recursively traversing the array until 0 is found. This can be seen in listing 4.1.10. This approach is similar to the intuition of string termination in *C* where the null character (`'\0'`) is used to tell the program where a string terminates. This implementation creates a chain of dependencies where the **map** is sequentially run before the **concat** and **loop** which results in a work complexity of $\mathcal{O}(\mathcal{W}_{map}(n,1)) + \mathcal{O}(\mathcal{W}_{concat}(n)) + \mathcal{O}(n)$, and equally for span.

```
let size = \arr.
    let arr' = map (\x.¬ 0) 0 arr in
        if (arr' = []) then 0
        else let arr'' = concat arr' [0] in
            fix (\counter . \i.
               if arr''[i] = 0 then i
               else (counter (i´+´1))
            ) 0
    in ...
```

Listing 4.1.10: Size implemented using **map**, **concat**, and recursion.

Since, in Futhark, size parameters are an essential part of the run-time, the cost of **size** is $\mathcal{O}(1)$ [31]. However, these size parameters are not part of ButF which makes **size** more complicated. In the translation of ButF (Eπ) the size of an array is defined in the construct similar to Futhark's run-time in the form of the channel *len*. This illustrates how the size of an array can be kept alongside the array which keeps the work and span of **size** at $\mathcal{O}(1)$.

4.1.6 Iota

Iota is besides **concat** one of two ways to define variable length arrays. This is useful in a variety of scenarios when building other primitives, for example, **size** can help create an array with the same length as the input array, where the array's values correspond to their indices, i.e. $[0,1,2,\ldots,n-1]$.

Iota can be built using **concat**, by iterating with recursion and concatenating the array elements one by one. This implementation can be seen in listing 4.1.11. Note that **loop** is equivalent to the recursive definition, but is used for readability.

This implementation using **loop** has a work of $\mathcal{O}(n \cdot \mathcal{W}_{concat}(n))$ and span of $\mathcal{O}(n \cdot \mathcal{S}_{concat}(n))$ since **loop** is sequential.

```
let iota = \n. (
    loop i = [] for x < n do concat i [x])
) in ...
```

Listing 4.1.11: Iota created from **concat** and **loop**.

```
let iota = fix (\f. \n.
    if (n = 0) then
        []
    else
        let h = / n 2
        in let h_iota = iota h
        in let res = concat h_iota (map (+ h) h_iota)
        in if (= n (+ h h)) then
              res
          else
              concat res [n-1]
) in ...
```

Listing 4.1.12: Iota using potentially parallel operators **concat** and **map**.

Alternatively, **iota** can be implemented by calling **iota** recursively on $n/2$ instead of $n-1$. This can be seen in listing 4.1.12, where **iota** is called recursively on half the input. We observe that if n is even, then the second half of the scan is the first half with $n/2$ added to each element. This observation can be used to reduce the work, by only calling **iota** once recursively. If n is odd, then the result (res) is $[0,1,2,\ldots,n-2]$, while we would expect it to end at $n-1$. In this case, $n-1$ is appended to the end. The complexity of this **iota** implementation can be expressed recursively as seen in equation (4.4).

$$\begin{aligned}\mathcal{W}_{iota}(n) &= \mathcal{W}_{iota}(n/2) + \mathcal{O}(\mathcal{W}_{concat}(n) + \mathcal{W}_{map}(n,1)) \\ \mathcal{S}_{iota}(n) &= \mathcal{S}_{iota}(n/2) + \mathcal{O}(\mathcal{S}_{concat}(n) + \mathcal{S}_{map}(n,1))\end{aligned} \tag{4.4}$$

Lastly, having **iota** as a primitive is assumed to have the same asymptotic complexity as FUTHARK with a work of $\mathcal{O}(n)$ and a span of $\mathcal{O}(1)$ [32]. With the recursive implementation, the minimum span is $log_2(n)$ in the best case, where **concat** and **map** have constant spans. This makes **iota** as a primitive more desirable as it is more effective than the derived implementations.

4.1.7 Concat

Concat is the second way to create arrays of arbitrary length, in that it creates an array by combining two other arrays.

It can be constructed by using **iota**, **size**, and **map** by creating a new array via **iota** of the combined length of the two arrays and mapping over this array. The mapping function then selects elements of either input array based on the index (from **iota**). The implementation of this can be seen in listing 4.1.13.

This requires that the arrays are applied to the function, which is then propagated to each element of the resulting array, resulting in a lot of duplicate arrays in the runtime. The asymptotic complexity of this **concat** depends on **map**, **size**, and **iota**. Thus the work is $\mathcal{O}(\mathcal{W}_{map}(n,1) + \mathcal{W}_{iota}(n) + \mathcal{W}_{size}(n))$ and the span is $\mathcal{O}(\mathcal{S}_{map}(n,1) + \mathcal{S}_{iota}(n) + \mathcal{S}_{size}(n))$.

```
let concat = (\arr. \brr.
   let s = size arr
   in let choose = (\i. if (i `<=` s)
                         then arr[i]
                         else brr[+ s i])
   in map choose (iota (+ s (size brr)))
) in ...
```

Listing 4.1.13: An implementation of array concatenation functionality using **map**, **size**, and **iota**.

If **concat** is chosen as a primitive the asymptotic work and span can be assumed to be the same as **concat** in FUTHARK, in which case the work is $\mathcal{O}(n)$ and the span is $\mathcal{O}(1)$ [32].

4.1.8 Selecting Primitives

When selecting the set of primitives, it is necessary that all derived primitives which are not in the set can be derived from the set of primitives. To prove the correctness of the translation, the set of primitive candidates is chosen based on the ease of proof. However, these implementations should also ideally have the exact same asymptotic complexity as the primitive.

To sum up the complexities and provide an overview table 4.1 is used to help select the set of primitives.

	Implementations		
Map	**scan**[†]: $\mathcal{S}$: $\mathcal{O}(\mathcal{S}_{scan}(n,\mathcal{S}_f))$ $\mathcal{W}$: $\mathcal{O}(\mathcal{W}_{scan}(n,\mathcal{W}_f))$	**size**, **concat**: $\mathcal{S}$: $2\cdot\mathcal{S}_{map}(n,\mathcal{S}_f)+\mathcal{O}(\mathcal{S}_{size}(n)+\mathcal{S}_{concat}(n))$ $\mathcal{W}$: $2\cdot\mathcal{W}_{map}(n,\mathcal{W}_f)+\mathcal{O}(\mathcal{W}_{size}(n)+\mathcal{W}_{concat}(n))$	Primitive: $\mathcal{S}$: $\mathcal{O}(\mathcal{S}_f)$ $\mathcal{W}$: $\mathcal{O}(n\cdot\mathcal{W}_f)$
Reduce	**size**: $\mathcal{S}$: $\mathcal{O}(\mathcal{S}_{size}(n)+n\cdot\mathcal{S}_f)$ $\mathcal{W}$: $\mathcal{O}(\mathcal{W}_{size}(n)+n\cdot\mathcal{W}_f)$	**scan**, **size**: $\mathcal{S}$: $\mathcal{O}(\mathcal{S}_{scan}(n,\mathcal{S}_f)+\mathcal{S}_{size}(n))$ $\mathcal{W}$: $\mathcal{O}(\mathcal{W}_{scan}(n,\mathcal{W}_f)+\mathcal{W}_{size(n)})$	Primitive: $\mathcal{S}$: $\mathcal{O}(log_2(n)+\mathcal{S}_f)$ $\mathcal{W}$: $\mathcal{O}(n\cdot\mathcal{W}_f)$
Scan	**map**, **size**, **iota** : $\mathcal{S}$: $\mathcal{S}_{scan}(n/2,\mathcal{S}_f)+\mathcal{O}(\mathcal{S}_{iota}(n)+\mathcal{S}_{map}(n,\mathcal{S}_f)+\mathcal{S}_{size}(n))$ $\mathcal{W}$: $\mathcal{W}_{scan}(n/2,\mathcal{W}_f)+\mathcal{O}(\mathcal{W}_{iota}(n)+\mathcal{W}_{map}(n,\mathcal{W}_f)+\mathcal{W}_{size}(n))$		Primitive: $\mathcal{S}$: $\mathcal{O}(log_2(n)+\mathcal{S}_f)$ $\mathcal{W}$: $\mathcal{O}(n\cdot\mathcal{W}_f)$
Size	**reduce**[†]: $\mathcal{S}$: $\mathcal{O}(\mathcal{S}_{reduce}(n,1))$ $\mathcal{W}$: $\mathcal{O}(\mathcal{W}_{reduce}(n,1))$	**scan**, **concat**: $\mathcal{S}$: $\mathcal{O}(\mathcal{S}_{scan}(n,1)+\mathcal{S}_{concat}(n)+n)$ $\mathcal{W}$: $\mathcal{O}(\mathcal{W}_{scan}(n,1)+\mathcal{W}_{concat}(n)+n)$	Primitive: $\mathcal{S}$: $\mathcal{O}(1)$ $\mathcal{W}$: $\mathcal{O}(1)$
Iota	**concat**: $\mathcal{S}$: $\mathcal{O}(n\cdot\mathcal{S}_{concat}(n))$ $\mathcal{W}$: $\mathcal{O}(n\cdot\mathcal{W}_{concat}(n))$	**map**, **concat**: $\mathcal{S}$: $\mathcal{S}_{iota}(n/2)+\mathcal{O}(\mathcal{S}_{concat}(n)+\mathcal{S}_{map}(n,1))$ $\mathcal{W}$: $\mathcal{W}_{iota}(n/2)+\mathcal{O}(\mathcal{W}_{concat}(n)+\mathcal{W}_{map}(n,1))$	Primitive: $\mathcal{S}$: $\mathcal{O}(1)$ $\mathcal{W}$: $\mathcal{O}(n)$
Concat	**map**, **size**, **iota**: $\mathcal{S}$: $\mathcal{O}(\mathcal{S}_{map}(n,1)+\mathcal{S}_{iota}(n)+\mathcal{S}_{size}(n))$ $\mathcal{W}$: $\mathcal{O}(\mathcal{W}_{map}(n,1)+\mathcal{W}_{iota}(n)+\mathcal{W}_{size}(n))$		Primitive: $\mathcal{S}$: $\mathcal{O}(1)$ $\mathcal{W}$: $\mathcal{O}(n)$

Table 4.1: A comparison table for an overview of the different work and span costs of the primitive candidates. [†]The call to the SOAC is with an illegal operator, ie. an operator that is not associative.

Size and Concat

A simple primitive set, that would allow us to construct all primitives is **size** and **concat**. The inferences necessary to demonstrate this can be seen in figure 4.1. The figure shows each step when deriving the full set of operators, starting at **size** and **concat**. The underline is used to denote the operator in question, for example in the first line where only **size** is used to derive **reduce**.

$$\{\underline{size}, concat\} \rightarrow \{\underline{reduce}, size, concat\}$$
$$\{reduce, \underline{size}, \underline{concat}\} \rightarrow \{\underline{map}, reduce, size, concat\}$$
$$\{\underline{map}, reduce, size, \underline{concat}\} \rightarrow \{map, reduce, size, \underline{iota}, concat\}$$
$$\{\underline{map}, reduce, size, \underline{iota}, concat\} \rightarrow \{map, reduce, \underline{scan}, size, iota, concat\}$$

Figure 4.1: Deriving all operators with just **size** and **concat** as primitive.

There is, however, one problem with this set of primitives which is the lack of concurrency in the primitives. As neither the **concat** nor **size** implementation has concurrency, the implementations using only these primitives cannot contain any concurrency. The span of the derived SOACs would likely be worse than that of Futhark. As such, any set of primitives that do not contain any of the three concurrent primitives (**map**, **reduce**, and **scan**) is not a good candidate for a primitive set.

Reduce and Concat

Another small and valid set of primitives is **reduce** and **concat**. With these as primitives, there is a form of concurrency due to the **reduce**.

$$\begin{aligned}
&\{\underline{reduce}, concat\} \to \{reduce, \underline{size}, concat\}\\
&\{reduce, \underline{size}, \underline{concat}\} \to \{\underline{map}, reduce, size, concat\}\\
&\{\underline{map}, reduce, size, \underline{concat}\} \to \{map, reduce, size, \underline{iota}, concat\}\\
&\{\underline{map}, reduce, size, \underline{iota}, concat\} \to \{map, reduce, \underline{scan}, size, iota, concat\}
\end{aligned}$$

Figure 4.2: The inference of all functions based on **concat** and **reduce** as primitives.

Figure 4.2 shows the inference focused on the parallel **iota** implementation. To infer **scan** with this primitive set, **map** and **iota** are needed since its the only available implementation of **scan**, This means **scan** can not be used to make **map**. It also follows that **size** is needed before **map** and therefore before **scan** as well. This then again means there is only one possible implementation for **size** too. However, there is a choice for **iota** with either a linear implementation with **concat** or a possibly sub-linear implementation whose complexity relies on the **concat** and **map**. The more concurrent implementation using both **concat** and **map** has a better span in this case due to the $\mathcal{O}(1)$ span of **concat**, which propagates to **size** and afterward to **map**. There is, however, a significant work overhead. With this, the complexities of the inferred implementations then become as seen in equation (4.5).

$$\begin{aligned}
&\mathcal{S}_{size}(n) = \mathcal{O}(\log_2(n)) && \mathcal{W}_{size}(n)) = \mathcal{O}(n)\\
&\mathcal{S}_{map}(n) = \mathcal{O}(\log_2(n) + \mathcal{S}_f) && \mathcal{W}_{map}(n) = \mathcal{O}(n \cdot \log_2(n) + n \cdot \mathcal{W}_f)\\
&\mathcal{S}_{iota}(n) = \mathcal{O}(\log_2(n)) && \mathcal{W}_{iota}(n) = \mathcal{O}(n \cdot \log_2(n))\\
&\mathcal{S}_{scan}(n) = \mathcal{O}(\log_2(n) + \mathcal{S}_f) && \mathcal{W}_{scan}(n) = \mathcal{O}(n \cdot \log_2(n) + n \cdot \mathcal{W}_f)
\end{aligned} \tag{4.5}$$

This shows reasonable complexities for the functions, however, **map** having a logarithmic span is not optimal compared to FUTHARK. Furthermore, **reduce** is assumed to not be the easiest operation to prove. Additionally both **size** and **iota** suffer from implementations with sub-optimal performance compared to FUTHARK.

Scan, Size, and Iota

To avoid the higher work and span of **size** and **iota**, they can instead be chosen as primitives. This would increase the number of primitives, however, due to the assumed simplicity of **size** and **iota** this is deemed an acceptable increase in proof difficulty. Using these together with **scan**, it is possible to infer the rest of the primitives, which can be seen in figure 4.4.

$$
\begin{aligned}
&\{\underline{scan}, size, iota\} \rightarrow \{\underline{map}, scan, size, iota\} \\
&\{map, \underline{scan}, \underline{size}, iota\} \rightarrow \{map, \underline{reduce}, scan, size, iota\} \\
&\{\underline{map}, reduce, scan, \underline{size}, \underline{iota}\} \rightarrow \{map, reduce, scan, size, iota, \underline{concat}\}
\end{aligned}
$$

Figure 4.3: Deriving operators from primitives **scan**, **size**, and **iota**.

One problem with this is that it relies on $\{scan\} \rightarrow \{map\}$ which restricts the **scan** to being left-associative. This comes with a cost to the concurrency since this requires all the function applications to happen sequentially.

We cannot implement **map** with **concat** and **size**, since **map** is required to implement **concat**. This is a problem, due to it removing the parallelism from **scan**, and in extension the primitive set. Thus **scan** is replaced with **map**, which can be used to efficiently derive **scan**.

Map, Size, and Iota

With **map**, **size**, and **iota** all other primitives can be constructed as seen in figure 4.4.

$$
\begin{aligned}
&\{\underline{map}, \underline{size}, iota\} \rightarrow \{map, size, iota, \underline{concat}\} \\
&\{\underline{map}, \underline{size}, \underline{iota}, concat\} \rightarrow \{map, \underline{scan}, size, iota, concat\} \\
&\{map, \underline{scan}, \underline{size}, iota, concat\} \rightarrow \{map, \underline{reduce}, scan, size, iota, concat\}
\end{aligned}
$$

Figure 4.4: Deriving operators from primitives **map**, **size**, and **iota**. Note how this is similar to figure 4.3, but **scan** is derived from **map** instead of the reverse.

We can now show how the asymptotic work and span complexities of the derived **concat**, **scan**, and **reduce** compare to FUTHARK. Equation (4.6) shows the work and span of **concat** as a derived operator, assuming that **size**, **map**, and **iota** inherit the complexities of FUTHARK.

$$
\begin{aligned}
\mathcal{S}_{concat}(n) &= \mathcal{O}(1) + \mathcal{O}(1) + \mathcal{O}(1) = \mathcal{O}(1) \\
\mathcal{W}_{concat}(n) &= \mathcal{O}(n \cdot 1) + \mathcal{O}(n) + \mathcal{O}(1) = \mathcal{O}(n)
\end{aligned}
\tag{4.6}
$$

The complexity of the derived **scan** operator is more complicated in that it is defined recursively. In this case, we utilize *the master theorem*, which can be used to find asymptotic bounds of a large collection of recursive functions [33]. It assumes that the function in question is constant for all sufficiently small inputs. We know that the **scan** in equation (4.3) has constant work and span in the base case where the input array has a length of less than two.

$$\begin{aligned}
\mathcal{S}_{scan}(n,\mathcal{S}_f) &= \mathcal{S}_{scan}(n/2,\mathcal{S}_f) + \mathcal{O}(1) + O(\mathcal{S}_f) + \mathcal{O}(1) \\
&= \mathcal{S}_{scan}(n/2,\mathcal{S}_f) + \mathcal{O}(1) \\
&= \mathcal{O}(\log_2 n) && \text{(by \textit{master theorem})} \\
\mathcal{W}_{scan}(n,\mathcal{W}_f) &= \mathcal{W}_{scan}(n/2,\mathcal{W}_f) + \mathcal{O}(n) + \mathcal{O}(n\cdot\mathcal{W}_f) + \mathcal{O}(1) \\
\mathcal{W}_{scan}(n,\mathcal{W}_f) &= \mathcal{W}_{scan}(n/2,\mathcal{W}_f) + \mathcal{O}(n\cdot\mathcal{W}_f)
\end{aligned}$$

Assuming that $\mathcal{S}_f$ does not depend on n, the complexity of **scan** is rewritten to the recursive call and $\mathcal{O}(1)$. $\mathcal{S}_{scan}(n,\mathcal{S}_f)$ is then on the form $T(n) = a\cdot T(n/b)+f(n)$ where $a = 1$, $b = 2$, and $f(n) = \mathcal{O}(1)$. In that $f(n) = \Omega(1)$, it must also hold that $f(n) = \Theta(n^{log_b a}) = \Theta(1)$, and thus, by *the master theorem*, $T(n) = \Theta(\log_2 n)$.

Applying *the master theorem* on the work of **scan** requires a lower bound on the work of **scan**, **size**, and **iota**, which is unknown. However, we believe that the work is $\mathcal{O}(n\cdot\mathcal{W}_f+n)$. This can be shown inductively, with explicit constants $a,b > 0$ instead of $\mathcal{O}$

$$\mathcal{W}_{scan}(n,\mathcal{W}_f) = \begin{cases} a & n < 2 \\ \mathcal{W}_{scan}(\lfloor n/2\rfloor,\mathcal{W}_f) + b\cdot n\cdot\mathcal{W}_f & n \geq 2 \end{cases} \tag{4.7}$$

We must then show that $\mathcal{W}_{scan}(n,\mathcal{W}_f) \leq c\cdot(n+\mathcal{W}_f\cdot n)$ for some $c > 0$. In the case where $n = 1$, we get $a \leq c(1+\mathcal{W}_f)$, which holds when $a \leq c$ assuming $\mathcal{W}_f \geq 0$. In the case where $n > 1$ the induction hypothesis can be substituted.

$$\begin{aligned}
\mathcal{W}_{scan}(n/2,\mathcal{W}_f) + b\cdot n + \mathcal{S}_f\cdot n &\leq \overbrace{c\cdot(\lfloor n/2\rfloor + \mathcal{W}_f\cdot\lfloor n/2\rfloor)}^{\text{subst}} + b\cdot n\cdot\mathcal{W}_f \\
&\leq c\cdot(n/2+\mathcal{W}_f\cdot n/2) + b\cdot n\cdot\mathcal{W}_f \\
&= \frac{1}{2}\cdot c\cdot n + \frac{1}{2}\cdot c\cdot\mathcal{W}_f\cdot n + b\cdot n\cdot\mathcal{W}_f \\
&\leq \frac{1}{2}\cdot c\cdot n + c\cdot\mathcal{W}_f\cdot n && (c \geq 2\cdot b) \\
&\leq c(n+\mathcal{W}_f\cdot n)
\end{aligned}$$

Lastly, **reduce** can be constructed with **scan** and **size**. This results in the work and span shown in equation (4.8).

$$\begin{aligned}
\mathcal{S}_{reduce}(n,\mathcal{S}_f) &= \mathcal{O}(\log_2 n + \mathcal{S}_f) + \mathcal{O}(1) = \mathcal{O}(\log_2 n + \mathcal{S}_f) \\
\mathcal{W}_{reduce}(n,\mathcal{W}_f) &= \mathcal{O}(n + n\cdot\mathcal{W}_f) + \mathcal{O}(1) = \mathcal{O}(n)
\end{aligned} \tag{4.8}$$

Notice how the upper bound on asymptotic work and span of the derived operators shown are matching the ones achieved by FUTHARK. **Size**, **iota**, and **map** are, therefore, the three operators chosen as primitives, while the others are derived.

4.2 Administrative Reductions

When translating a BUTF expression several channels are introduced to transmit values in the translated program. Due to the differences between the languages, a single reduction in BUTF can be matched by one or more reductions in the translated process. Some of these reductions are only necessary to move values to the right processes and do not correspond to any reduction in BUTF. These reductions are considered *administrative reductions*. To differentiate between these, a new reduction rule is created. This can be seen in figure 4.5, and it allows for a distinction between administrative ($\xrightarrow{\circ}$) and important reductions ($\xrightarrow{\bullet}$). The set of administrative reductions is defined as the complement set of the important reductions ($\overline{\{\bullet\}}$). In the case where it can be either, $\xrightarrow{s}$ is used where $s \in \{\circ, \bullet\}$.

The additional syntax $\bullet A$ is introduced to explicitly annotate an action as important. For example when writing $\bullet\overline{a}\langle 5\rangle.P$, sending on a is done as an important reduction. Figure 4.5 defines the IMP reduction allowing important reductions on such processes.

$$\textsc{Adm}\ \frac{A \rightarrow A'}{A \xrightarrow{\circ} A'} \qquad \textsc{Imp}\ \frac{A \xrightarrow{\circ} A'}{\bullet A \xrightarrow{\bullet} A'}$$

Figure 4.5: Labeled semantics to define important ($\xrightarrow{\bullet}$) and administrative reductions ($\xrightarrow{\circ}$) in Eπ.

The important annotations are added in the translation such that they match the reductions in BUTF. This can be seen in figure 4.6. With this notation, some of the translations change slightly. Note that not all translations are annotated, as not all translations correspond to a reduction in BUTF. Application has an annotation placed when sending on the function channel (f), as this corresponds to a function application in BUTF.

Let for values, has an annotation placed when receiving a value, which is used to substitute the variable, and similarly let for tuples has an annotation when extracting and substituting the values from the tuple. Both of these correspond to the reduction on let in BUTF. Indexing is annotated on the receiving of the value. If is annotated when checking the condition, as this is the same as the if reduction in BUTF. Finally, map has an annotation where it sends to the function channel, which is necessary since the translation of map does not make use of the normal application rule.

$$
\begin{array}{rll}
[\![\textbf{let } x = e_1 \textbf{ in } e_2]\!]_o \overset{\text{def}}{=} & \nu o_1.([\![e_1]\!]_{o_1} \mid \bullet o_1(x).[\![e_2]\!]_o) & \text{Let-Val} \\
[\![\textbf{let } (x_0,\dots,x_n) = e_1 \textbf{ in } e_2]\!]_o \overset{\text{def}}{=} & \nu o_1.([\![e_1]\!]_{o_1} \mid o_1(h).\bullet h \cdot h(x_0,\dots,x_n).[\![e_2]\!]_o) & \text{Let-Tup} \\
[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]_o \overset{\text{def}}{=} & \nu o_1.([\![e_1]\!]_{o_1} \mid o_1(v).\bullet [v \neq 0]\ [\![e_2]\!]_o, [\![e_3]\!]_o) & \text{If}
\end{array}
$$

$$
\begin{array}{rll}
[\![e_1\ e_2]\!]_o \overset{\text{def}}{=} & \nu o_1.\nu o_2.([\![e_1]\!]_{o_1} \mid [\![e_2]\!]_{o_2} \mid o_1(f).o_2(x).\bullet \overline{f}\langle x,o\rangle) & \text{Application} \\
[\![e_1[e_2]]\!]_o \overset{\text{def}}{=} & \nu o_1.\nu o_2.([\![e_1]\!]_{o_1} \mid [\![e_2]\!]_{o_2} & \\
& \quad \mid o_1(h).o_2(i).[i \geq 0]\ \bullet h \cdot i(v).\overline{o}\langle v\rangle, \mathbf{0}) & \text{Indexing}
\end{array}
$$

$$
\begin{array}{rl}
\mathsf{Map}(\mathit{input}, f, \mathit{out}) \overset{\text{def}}{=} & !\mathit{input}(\mathit{index}, \mathit{value}).\nu r.(\bullet \overline{f}\langle \mathit{value}, r\rangle \mid \\
& \quad r(\mathit{value}').\overline{\mathit{out}}\langle \mathit{index}, \mathit{value}'\rangle)
\end{array}
$$

Figure 4.6: The annotated translations.

The distinction between important and administrative reductions is used to create a new weak administrative bisimilarity relation. Here, a single important reduction matches one reduction in ButF, and vice versa, whilst allowing arbitrary many administrative reductions. Here, we introduce the arrows $\overset{\bullet}{\Rightarrow} = \overset{\circ}{\to}^* \overset{\bullet}{\to} \overset{\circ}{\to}^*$ and $\overset{\circ}{\Rightarrow} = \overset{\circ}{\to}^*$. Notice how $\overset{\bullet}{\Rightarrow}$ requires at least one important reduction while $\overset{\circ}{\Rightarrow}$ allows zero or more administrative reductions.

Definition 4.2.1 (Weak Administrative Bisimulation) Let R be a binary relation between processes, such that R is symmetric. Then R is a *weak administrative bisimulation*

1. if $P \overset{\bullet}{\to} P'$ then there exists a Q' such that $Q \overset{\bullet}{\Rightarrow} Q'$ and $(P', Q') \in R$, and

2. if $P \overset{\circ}{\to} P'$ then $Q \overset{\circ}{\Rightarrow} Q'$ and $(P', Q') \in R$.

We relate two processes P and Q, i.e. $P \approx_a Q$ if there exists a weak administrative bisimulation R such that $(P, Q) \in R$.

Compared to the weak bisimulation defined in section 2.3.3, the administrative weak bisimulation is able to distinguish between processes. For example, a process that can take an important reduction, would not be weak administrative bisimilar with the $\mathbf{0}$ process. This makes it more useful since it allows for a more meaningful relation between processes. However, it does not distinguish between the outputs of the processes, such as $\overline{a}\langle 5\rangle$ and $\overline{a}\langle 6\rangle$. Since a translation should always produce the same result, a stricter relation is needed

4.3 Weak Barbed Bisimulation

To create a stricter relation, barbs are combined with the weak administrative bisimulation. The definition of this relation can be seen in definition 4.3.1.

Definition 4.3.1 (Weak Administrative Barbed Bisimulation) Let R be a binary relation between processes such that R is symmetric. Then R is a *weak administrative barbed bisimulation* if for all pairs $(P,Q) \in R$ the following holds

1. if $P \overset{\bullet}{\to} P'$ then there exists a Q' such that $Q \overset{\bullet}{\Rightarrow} Q'$ and $(P',Q') \in R$,
2. if $P \overset{\circ}{\to} P'$ then $Q \overset{\circ}{\Rightarrow} Q'$ and $(P',Q') \in R$,
3. for all contexts C, $(C[P],C[Q]) \in R$,
4. and for all prefixes α, if $P \downarrow_\alpha$ then $Q \overset{\circ}{\Rightarrow} \downarrow_\alpha$.

We relate two processes P and Q, i.e. $P \approx_a Q$ if there exists a weak administrative barbed bisimulation R such that $(P,Q) \in R$.

The following lemmas show some properties of $\approx_a$, such as the behavior of the **0** process. While not directly related to $\approx_a$, lemma WABB 1 describes how contexts can reduce, which is useful in following lemmas.

Lemma WABB 1 For any P and C, if Q exists such that $C[P] \xrightarrow{s} Q$, then one of the following holds:

1. C reduces alone, thus $Q = C'[P]$ with context C' such that $C[\mathbf{0}] \xrightarrow{s} C'[\mathbf{0}]$,
2. P reduces alone, thus $Q = C[P']$ with $P \xrightarrow{s} P'$, and
3. C and P interact, thus $Q = C'[P']$ for P' and C' such that O exists where $O \mid P \xrightarrow{s} O' \mid P'$, $C[P] \xrightarrow{s} C'[P']$, and $C[P] \equiv \nu\vec{a}.(O \mid P)$.

Proof Let us consider two main possibilities; P appears in $C[P]$ unguarded or guarded. If P is guarded, it cannot communicate with C nor take any reductions, meaning C can only reduce alone, i.e. the first case.

Otherwise, P is unguarded in $C[P]$, which can only happen if P is behind a combination of !, νa, or $Q \mid$ in C. Using structural congruence, we can rearrange these terms as follows for some other process O.

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

Note that O might contain P itself, or even $!P$. In the latter case that $C[P] = !\nu\vec{a}.(O \mid P)$, it can still be rearranged to the former case.

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

In that $C[P]$ and $\nu\vec{a}.(O \mid P)$ can take the same reductions, we know that Q is one of the following three cases.

- *Case* $\boxed{Q \equiv \nu\vec{a}.(O \mid P'), \quad P \xrightarrow{s} P'}$ In this case, the reduction was internal to P, and therefore $Q = C[P']$, thus falling into the second lemma case.

- *Case* $\boxed{Q \equiv \nu\vec{a}.(O' \mid P), \quad O \xrightarrow{s} O'}$ In this case, the reduction is internal to O, and $Q = C'[P]$ for some new context C', such that $C[P] \xrightarrow{s} C'[P]$. We know that P could not have influenced the reductions to $C'[P]$, and we can therefore state $C[\mathbf{0}] \xrightarrow{s} C'[\mathbf{0}]$. This, therefore, falls into the first lemma case.

- *Case* $\boxed{Q \equiv \nu\vec{a}.(O' \mid P'), \quad O \mid P \xrightarrow{s} O' \mid P'}$ Here, O and P both contributed to the reduction, which could only have happened with either the Comm or the Broad rules. Therefore, there exists a name or composite name (b) such that $P \downarrow_b \wedge Q \downarrow_{\overline{b}}$ or vice versa. We can write Q using a new context C', i.e. $Q = C'[P']$, thus letting this case fall into the third lemma case. □

Lemma WABB 2 states that if a process is bisimilar to $\mathbf{0}$ then this process will always be similar to $\mathbf{0}$, and can not be observed on any prefix α. This is useful when attempting to remove "garbage" processes, which are bisimilar with $\mathbf{0}$.

Lemma WABB 2 If $P \approx_a \mathbf{0}$, then for any P' where $P \overset{s}{\Rightarrow}^* P'$ it holds that $\forall \alpha. P' \not\downarrow_\alpha$ and $P' \approx_a \mathbf{0}$.

Proof We use $\rightarrow^n$ to denote exactly n reductions by either $\xrightarrow{\bullet}$ or $\xrightarrow{\circ}$, i.e. $\rightarrow^n = \xrightarrow{s_1}\xrightarrow{s_2} \ldots \xrightarrow{s_n}$ for $s_1, \ldots, s_n \in \{\bullet, \circ\}$. We then show by induction that for any $n \geq 0$ if $P \rightarrow^n P'$ then $\forall \alpha. P' \not\downarrow_\alpha$ and $P' \approx_a \mathbf{0}$.

Case $\boxed{n = 0}$ Here, $P' = P$ and thus we must show that $P \not\downarrow_\alpha$ for any prefix. Let us assume the contradictory case where there exists an α such that $P \downarrow_\alpha$. We know that there exists a weak administrative barbed bisimulation (R) such that $(P, \mathbf{0}) \in R$, thus from the fourth requirement, because $P \downarrow_\alpha$ then $\mathbf{0} \overset{\circ}{\Rightarrow}\downarrow_\alpha$. However, this is impossible in that $\mathbf{0} \not\rightarrow$ and $\mathbf{0} \not\downarrow_\alpha$.

Case $\boxed{n > 0}$ From the induction hypothesis we know if $P \rightarrow^{n-1} P'$ then $\forall \alpha. P' \not\downarrow_\alpha$ and $P' \approx_a \mathbf{0}$. We assume $P \rightarrow^{n-1} P'$ because otherwise the above holds trivially. Then we must show that if $P' \xrightarrow{s} P''$ then $\forall\alpha.P'' \not\updownarrow_{\alpha}$ and $P'' \dot{\approx}_a \mathbf{0}$. In that $P' \dot{\approx}_a \mathbf{0}$ we know that an R exists such that $(P', \mathbf{0}) \in R$ and R is an administrative barbed bisimulation. Since $\mathbf{0} \overset{\bullet}{\not\rightarrow}$, then $P' \overset{\bullet}{\not\rightarrow}$ however we know that $P' \xrightarrow{\circ} P''$, in that P'' exists. Therefore, because $(R', \mathbf{0}) \in R$, then $(P'', \mathbf{0}) \in R$, and thus $P'' \dot{\approx}_a \mathbf{0}$ and $\forall\alpha.P \not\updownarrow_{\alpha}$. □

Lemma WABB 3 states how "garbage processes" can not affect other processes, and can thus be removed. This would thus allow for a basic garbage collection, by removing all the unreachable processes.

Lemma WABB 3 If $P \dot{\approx}_a 0$, then for all Q it holds $P \mid Q \dot{\approx}_a Q$.

Proof Let R be the relation $R = \{(C[P \mid Q], C[Q]) \mid P, Q \in \mathcal{P}_s, C \in \mathcal{C}_s, P \dot{\approx}_a \mathbf{0}\}$. It is then demonstrated that R^s is a weak administrative barbed bisimulation.

Let us consider any pair $(C[P \mid Q], C[Q]) \in R^s$ against the requirement of weak administrative barbed bisimulation. By the first two requirements if $C[P \mid Q] \xrightarrow{s} O$ then $C[Q]$ should follow with $\overset{s}{\Rightarrow}$. By lemma WABB 1 O is one of either three cases.

- C' reduces alone, i.e. $O = C'[P \mid Q]$. Here $C[Q]$ can follow by $\xrightarrow{s}$.
- $P \mid Q$ reduces alone, i.e. $O = C[S]$, for $P \mid Q \xrightarrow{s} S$. Because P can not observed on any channels (lemma WABB 2), either P or Q reduced alone. Thus if $S = P \mid Q'$ where $Q \xrightarrow{s} Q'$ then $C[Q] \xrightarrow{s} C[Q']$, and $(C[P \mid Q'], C[Q']) \in R$.

  Else if $S = P' \mid Q$ where $P \xrightarrow{s} P'$, and because $P \dot{\approx}_a \mathbf{0}$, s must be $\circ$. Then $C[Q]$ can follow with no reductions. Because, by lemma WABB 2, $P' \dot{\approx}_a \mathbf{0}$ we know that $(C[P' \mid Q], C[Q]) \in R$.
- $P \mid Q$ and C both contribute to the reduction, i.e. there must exist an S such that $S \mid P \mid Q \xrightarrow{s} S' \mid P' \mid Q'$, S is a sub-process of C, and there exists C' which has S' as a sub-process. We know that P can only reduce by itself, and thus $P' = P$ and $S \mid Q \xrightarrow{s} S' \mid Q'$. Then $C[Q]$ must be able to follow with $C[Q] \xrightarrow{s} C'[Q]$.

The third requirement automatically holds by our selection of R. The fourth requirement requires that if $C[P \mid Q] \downarrow_{\alpha}$ then $Q \overset{\circ}{\Rightarrow}\downarrow_{\alpha}$. From lemma WABB 2 we know that P cannot be observed with any prefix, thus if we can observe $P \mid Q$ with a prefix α it must be because we can observe Q with α.

Then we consider pairs $(C[Q], C[P \mid Q]) \in R^s$. Again we enumerate the cases of lemma WABB 1.

- When C reduces alone, we can follow the same argumentation as before.
- Q reduces alone, and thus $O = C[Q']$ where $Q \xrightarrow{s} Q'$. Here, $C[P \mid Q]$ can follow to $C[P \mid Q']$.

- C and Q both contribute to the reduction, and we can then follow the same argumentation as before.

For the fourth requirement, we require that if $C[Q] \downarrow_\alpha$ then $C[P \mid Q] \downarrow_\alpha$, which holds. Having considered every pair in R^s, we know that the lemma holds. □

Lemma WABB 4 states that $\dot{\approx}_a$ is transitive. This is a common property to show with weak bisimulations, as this demonstrates that it is an equivalence relation.

Lemma WABB 4 For any three processes (P, Q, and S), if $P \dot{\approx}_a Q$ and $Q \dot{\approx}_a S$ then $P \dot{\approx}_a S$.

Proof We know that there exist two weak administrative barbed bisimulations (R_1 and R_2), such that $(P,Q) \in R_1$ and $(Q,S) \in R_2$. Then we can construct a witness for $P \dot{\approx}_a S$ as $R = \{(C[P],C[S]) \mid C \in \mathcal{C}, (P,Q_1) \in R_1, (Q_2,S) \in R_2, Q_1 = Q_2\}$.

We must then show that R^s is a weak administrative barbed bisimulation. First, we consider every pair $(P,S) \in R^s$. We know that there exists Q such that $(P,Q) \in R_1$ and $(Q,S) \in R_2$. If $P \xrightarrow{\bullet} P'$ then $Q \overset{\circ}{\Rightarrow} Q' \xrightarrow{\bullet}\overset{\circ}{\Rightarrow} Q''$ and $(P',Q'') \in R_1$. We know that $Q \xrightarrow{\circ} Q_1 \ldots Q_n \xrightarrow{\circ} Q'$, and each Q_i has a corresponding S_i where $(Q_i,S_i) \in R_2$. Therefore, $S \overset{\circ}{\Rightarrow} S'$ where $(Q',S') \in R_2$. Because $Q' \xrightarrow{\bullet} Q''$, S' can follow with $\overset{\bullet}{\Rightarrow} S''$ where $(Q'',S'') \in R_2$, and therefore $(P',S'') \in R$.

If $P \downarrow \alpha$ then S should follow with $S \overset{\circ}{\Rightarrow}\downarrow_\alpha$. We do know that $Q \overset{\circ}{\Rightarrow} Q' \downarrow_\alpha$, and then per the same argument as above $S \overset{\circ}{\Rightarrow} S'$ where $(Q',S') \in R_2$. Then $S' \overset{\circ}{\Rightarrow} S'' \downarrow_\alpha$, and thus $S \overset{\circ}{\Rightarrow} S' \overset{\circ}{\Rightarrow}\downarrow_\alpha$. The same can be shown for pairs $(S,P) \in R^s$.

Lastly, for pairs $(C[P],C[S])$ and $(C[S],C[P])$ in R^s, we know that Q exists such that $(C[P],C[Q]) \in R_1$ and $(C[Q],C[S]) \in R_2$. Then we can follow the same argument as above. □

Lemma WABB 5 For any two process processes P and Q, if S exists such that $P \dot{\approx}_a S$ and $S \xrightarrow{s} Q$ then $P \overset{s}{\Rightarrow} \dot{\approx}_a Q$

Proof Because $P \dot{\approx}_a S$ then there exists weak administrative barbed bisimulation R such that $(P,S) \in R$. Then per weak administrative barbed bisimulation, because $S \xrightarrow{s} Q$ then $P \overset{s}{\Rightarrow} P'$ and $(P',Q) \in R$. Then we know that $P' \dot{\approx}_a Q$, and $P \overset{s}{\Rightarrow} \dot{\approx}_a Q$.□

Lemma WABB 6 shows that if P reaches some process Q through administrative reductions, then P and Q are bisimilar.

Lemma WABB 6 For any P,Q then $P \dot{\approx}_a Q$ if $P \overset{\circ}{\Rightarrow} Q$ and for every P'' where $P \xrightarrow{s} P''$ it holds that either

1. $P'' \overset{\circ}{\Rightarrow} Q$, and P'' is closed, i.e. $\forall \alpha . P'' \not\downarrow_\alpha$, or

2. $Q \overset{s}{\Rightarrow} P''$.

Proof Here, we propose the relation $R = \{(C[P'], C[Q]) \mid P \overset{\circ}{\Rightarrow} P' \wedge Q \overset{\circ}{\not\Rightarrow} P', C \in \mathcal{C}_s\} \cup \{(C[Q'], C[Q']) \mid Q \overset{\bullet}{\Rightarrow}^{*} Q', C \in \mathcal{C}_s\}$, and show that R^s is a weak administrative barbed bisimulation.

First consider pairs $(P', Q), (Q, P') \in R^s$ where P' is some process where $P \overset{\circ}{\Rightarrow} P'$. If $P' \overset{\circ}{\to} P''$, then $P'' \overset{\circ}{\Rightarrow} Q$, in which case Q can follow with zero reductions and $(P'', Q) \in R$. If $P' \overset{\bullet}{\to} P''$ then per definition Q can follow with $Q \overset{\bullet}{\Rightarrow} P''$ and $(P'', P'') \in R$. We know that $P' \overset{\circ}{\Rightarrow} Q$ and therefore if $Q \overset{s}{\to} Q'$ then $P' \overset{\circ}{\Rightarrow}\overset{s}{\to} Q'$ and $(Q', Q') \in R$. Also if for some α, $Q \downarrow_\alpha$ we know that $P' \overset{\circ}{\Rightarrow}\ Q \downarrow_\alpha$.

Secondly we consider pairs $(C[P'], C[Q]) \in R^s$ where $P \overset{\circ}{\Rightarrow} P'$. If $C[P'] \overset{s}{\to} O$ then because P' is closed, O is either $C'[P']$ for $C[\mathbf{0}] \overset{s}{\to} C'[\mathbf{0}]$ or $C[P'']$ for $P' \overset{s}{\to} P''$. In the former case $C[Q]$ can follow with $\overset{s}{\to}$ and $(C'[P'], C'[Q]) \in R$, and in the latter case the previous argument is reused. We use the same argument for pairs $(C[Q], C[P']) \in R^s$ noting that $P \overset{\circ}{\Rightarrow} Q$.

Pairs $(C[Q'], C[Q']) \in R$ hold trivially. And thus because $(P, Q) \in R^s$ and R^s is a weak administrative barbed bisimulation, the lemma holds. □

4.4 Proving Correctness

With the new weak administrative barbed bisimulation and the smaller set of primitives in ButF, the next step is to prove the correctness of the translation by the notion of operational correspondence.

4.4.1 Relaxing Operational Correspondence

The operational correspondence of section 3.1 defines correctness through the idea that the translation should match reductions in ButF. However, with the current translation $[\![]\!]_o$ this is too strong, mainly due to two reasons.

1. Some translated expressions, such as arrays, can take administrative reduction steps after being translated, thus overshooting the goal $[\![e']\!]_o$ that the operational correspondence requires.

2. The translation introduces new replicated processes when creating arrays or functions, which never disappear. Such "leftovers" are not present in the goal ($[\![e']\!]_o$) and therefore make it impossible to achieve structural congruence with the goal.

The second reason can be addressed by allowing the translation to get close to $[\![e']\!]_o$ through the notion of weak administrative barbed bisimulation.

To facilitate a more concise operational correspondence, it is defined as a non-symmetric relation (R) between expressions and processes. This is inspired by the work of Amadio, Thomsen, and Thomsen, Milner [17, 15]. We introduce the $\gtrsim_{ok}$ relation, as the largest relation upholding the requirements for administrative operational correspondence. Then a proof via co-induction can be used, by finding a witness R which is a subset of $\gtrsim_{ok}$ [26].

Definition 4.4.1 (Administrative Operational Correspondence) Let R be a binary relation between an expression and a process. Then R is an *administrative operational correspondence* if $\forall (e,P) \in R$ it holds that

1. if $e \rightarrow e'$ then there $\exists P'$ such that $P \overset{\circ}{\Rightarrow} \overset{\bullet}{\rightarrow} \approx_a P'$ and $(e',P') \in R$, and
2. if $P \overset{\circ}{\Rightarrow} \overset{\bullet}{\rightarrow} P'$ then there $\exists e',Q$ such that $e \rightarrow e'$, $Q \approx_a P'$, and $(e',Q) \in R$.

We denote $e \gtrsim_{ok} P$ if there exists an operational correspondence relation R such that $(e,P) \in R$.

4.4.2 Correctness of Translation

To prove the correctness of the translation, all the translations of the chosen primitive language constructs are considered.

First, a set of lemmas are formed which are useful in regard to proving the translation. Second, operational correspondence is proven to hold for all translations. Third, rewrites are proven to not affect the operational correspondence. Lemma OK 1 and conjecture 1, combined, state that a translation of an expression sends on o if and only if it is a value.

Lemma OK 1 If some expression e is a value then $\exists P.[\![e]\!]_o \overset{\circ}{\Rightarrow} P \land P \downarrow_{\overline{o}}$

Proof We assume that e is a value, as otherwise, the lemma holds trivially. We introduce a notation $\mathcal{D}(e)$ for the depth of a value expression (e). If e is a number or an abstraction, then $\mathcal{D}(e) = 0$. However, if e is a tuple or array with elements e_0 to e_m, then $\mathcal{D}(e) = \max_{i\in[0..m]}(\mathcal{D}(e_i)) + 1$.

By induction on $\mathcal{D}(e)$ we show that the lemma holds for all e. In the base case $\mathcal{D}(e) = 0$, and thus e is either a number or abstraction. From the translation of either number or abstraction, we know that $[\![e]\!]_o \downarrow_{\overline{o}}$, upholding the lemma for e.

In the inductive case, where $\mathcal{D}(e) > 0$, e must be either a tuple or array with elements e_0 to e_m. Here, the lemma holds for all e' where $\mathcal{D}(e') < \mathcal{D}(e)$, and in extension e_0 to e_n. If e is a tuple, then we can take reductions such that $[\![e_0]\!]_{o_0}$ to $[\![e_m]\!]_{o_m}$ all send on channels $o_0,\ldots,o_m$. Then $\overline{o}\langle h\rangle$ is unguarded.

For array, after it has sent on channels o_0 to o_m, it can then receive on *done* because it has sent $m+1$ values. Then $\overline{o}\langle handle\rangle$ is unguarded. □

Conjecture 1 If for some expression e, $\exists P.[\![e]\!]_o \overset{\circ}{\Rightarrow} P \wedge P\downarrow_{\overline{o}}$ then $e \in \mathcal{V}$ or $e \rightarrow\!\!\!\!\!\!\neg \in \mathcal{V}$.

Is is assumed that this conjecture holds, as it relies on lemma OK 1. The conjecture could be proven by iterating over all the translations, but this has been left out due to time constraints.

Lemma OK 2 states that a replicated process that listens on a handle (a), can be moved out of any context (C) given that $a \notin fn(C) \cup bn(C)$.

Lemma OK 2 Let a be a name and C be some context such that $a \notin fn(C) \cup bn(C)$. Also let P and Q be any process where $fn(P) \cap bn(C) = \varnothing$

Then $\nu a.C[P \mid Q] \dot{\approx}_a \nu a.(P \mid C[Q])$.

Proof We introduce the relation R:

$$\begin{aligned} R = \{ & (K[\nu a.C[P \mid Q], K[\nu a.(P \mid C[Q])]) \mid K, C \in \mathcal{C}_s, P, Q \in \mathcal{P}_s, \\ & P = !a(\dots).P_b \vee P = !\overline{a}\langle\dots\rangle.P_b, fn(P) \cap bn(C) = \varnothing, a \notin fn(C) \cup bn(C)\} \end{aligned}$$

By showing that R^s is a weak administrative barbed bisimulation, we know that $\nu a.C[P \mid Q] \dot{\approx}_a \nu a.(P \mid C[Q])$.

Let us consider any pair $(K[\nu a.C[P \mid Q], K[\nu a.(P \mid C[Q])]) \in R^s$. If $K[\nu a.(P \mid C[Q])] \overset{s}{\rightarrow} O$, then there exists three cases for O by lemma WABB 1.

- If K reduces alone such that $O = K'[\nu a.(P \mid C[Q])]$ then $K[\nu a.C[P \mid Q]]$ can follow and $(K'[\nu a.C[P \mid Q]], K'[\nu a.(P \mid C[Q])]) \in R$.
- If $\nu a.(P \mid C[Q])$ reduce together with the context K, then there exists a process S such that $S \mid \nu a.(P \mid C[Q]) \overset{s}{\rightarrow} S' \mid \nu a.(P \mid Q')$. We know that P is preserved in that it only communicates on a, which is restricted from S. Therefore we know that $S \mid \nu a.C[P \mid Q] \overset{s}{\rightarrow}$.
- If $\nu a.(P \mid C[Q])$ reduces alone, then we can identify the following cases
  - If $C[Q]$ reduces according to lemma WABB 1, in which $\nu a.C[P \mid Q]$ can always follow.
  - If Q is unguarded in C and P reduces together with Q over channel a, i.e. $P \mid C[Q] \overset{s}{\rightarrow} P \mid P_b' \mid C[Q']$, where P_b' is a process created by P. Through structural congruence, we can move P_b' into C such that $P_b' \mid C[Q] \equiv C[P_b' \mid Q]$. In that Q and in extension the hole in C is unguarded, we know that $C[P \mid Q] \overset{s}{\rightarrow} C[P \mid P_b' \mid Q']$. $P_b' \mid Q'$ is a normal process and thus $(K[\nu a.C[P \mid P_b' \mid Q'], K[\nu a.(P \mid C[P_b' \mid Q'])]) \in R$.

Instead, if $K[\nu a.C[P \mid Q]]$ can be reduced, we can follow the same argument.

If the restriction on a was not in-place, then requirement four would be broken, in that $C[P \mid Q]$ might not be observable on either a or $\overline{a}$. However, with the restriction in-place, $K[\nu a.P \mid C[Q]]$ and $K[\nu a.C[P \mid Q]]$ cannot differ in the channels they expose. □

Conjecture 2 shows that substitution is preserved by the translation.

Conjecture 2 For $e_1 \in \mathcal{V}$ and $e_2 \in \mathcal{B}$, then we have that

1. if e_1 is a number (n) then $[\![e_2]\!]_o\{{}^{n}/_{x}\} \approx_a [\![e_2\{x := n\}]\!]_o$,
2. or if e_1 is an abstraction, tuple, or array then there exists a channel (h) and $\nu h.(Q \mid [\![e_2]\!]_o\{{}^{x}/_{h}\}) \approx_a [\![e_2\{x := e_1\}]\!]_o$. Here, Q is $[\![e_1]\!]_o$ after sending h on o, i.e. $[\![e_1]\!]_o \mid o(x).P \overset{\circ}{\Rightarrow}\overset{\circ}{\rightarrow} \nu h.(Q \mid P\{{}^{h}/_{x}\})$.

Theorem 1 states that the translations of any rewritten expression are weak administrative barbed bisimilar to the original. This ensures that the value sent on o is correct.

Theorem 1 If $e \rightharpoondown e'$ then $[\![e]\!]_o \approx_a [\![e']\!]_o$.

PROOF To show that this holds, we consider every $\rightharpoondown$ reduction in the BUTF semantics.

Binary operation $\boxed{e = e_1 \text{`}e_2\text{`}\, e_3}$ By the translation $[\![e_1 \text{`}e_2\text{`}\, e_3]\!] = [\![e_2 e_1 e_3]\!]$ and thus $[\![e]\!]_o = [\![e']\!]_o$ and therefore they are also weak administrative barbed bisimilar.

Currying of cf $\boxed{e = cf}$ Each cf ($\odot$, $\triangleright$, **size**, **iota**, **map**) has a matching abstraction translation in Eπ.

Binop and Unary Operator When $e = \odot\{x,y\}$ or $e = \triangleright \{x\}$. Here, $[\![e]\!]_o$ is either $\overline{o}\langle x \odot y\rangle$ or $\overline{o}\langle \triangleright x\rangle$ respectively. In that $\odot$ and $\triangleright$ behave identically in BUTF and Eπ, $[\![e]\!]_o$ and $[\![e']\!]_o$ sends the same value on o.

Size $\boxed{e = size\{v\}}$ The semantics only allows $e \rightharpoondown e'$ if $v = [v_0,\ldots,v_{n-1}]$. Then we must show that $[\![e]\!]_o \stackrel{.}{\approx}_a \overline{o}\langle n\rangle$, and thus we propose $R = \{(C[P], C[\overline{o}\langle n\rangle]) \mid P \in \mathcal{P}_s \mid C \in \mathcal{C}, [\![e]\!]_o \overset{\circ}{\Rightarrow} P\} \cup \{(C[Z], C[\mathbf{0}]) \mid C \in \mathcal{C}, Z \in \mathcal{P}_s, Z \approx_a \mathbf{0}\}$, such that P^s is a witness.

For the pair $(C[P], C[\overline{o}\langle n\rangle]) \in R^s$, we must show that the weak administrative barbed bisimulation requirements are satisfied. If $C[P] \xrightarrow{s} Q$ then by lemma WABB 1, this can happen in three ways. If $Q = C[P']$ where $P \xrightarrow{s} P'$, then $s = \circ$ because $[\![e]\!]_o$ can never take important reductions, and therefore $(C[P'], C[\overline{o}\langle n\rangle]) \in R$. Since $[\![e]\!]_o = \nu o_1.(\overline{o_1}\langle x\rangle \mid o_1(h).h(\_, len).\overline{o}\langle len\rangle)$ it can easily be observed, that no important reduction can happen. If $Q = C'[P]$ where $C[\mathbf{0}] \xrightarrow{s} C'[\mathbf{0}]$, then $(C'[P], C'[\overline{o}\langle n\rangle]) \in R$. If C and P reduce together, then $P \downarrow_{\overline{o}}$, meaning $C[P] = C[\nu h.S \mid \overline{o}\langle n\rangle]$, where S is the array leftovers of $[\![v]\!]_o$, where $\nu h.S \stackrel{.}{\approx}_a \mathbf{0}$. Then $Q = C'[\nu h.S]$, and $C[\overline{o}\langle n\rangle]$ can follow the same reduction and $(C'[\nu h.S], C'[\mathbf{0}]) \in R$.

We also know by the definition of $[\![e]\!]_o$ that $\forall P, \alpha. [\![e]\!]_o \overset{\circ}{\Rightarrow} P \land P \downarrow_\alpha \implies \alpha = \overline{o}$, i.e. it is only ever observable on o. This matches $\overline{o}\langle n\rangle$, and therefore if $C[P] \downarrow_\alpha$ then $C[\overline{o}\langle n\rangle] \downarrow_\alpha$.

Pairs $(C[\overline{o}\langle n\rangle], C[P]) \in R^s$, can be shown in a similar fashion.

Map $\boxed{e = map\{v_f, v_a\}}$ We assume that $v_f = \lambda x.e_b$, and $v_a = [v_0, \ldots, v_{n-1}]$, in that $e \rightharpoondown e'$. First, we consider the translation of $[\![e]\!]_o$ as seen in equation (4.9).

$$[\![e]\!]_o = \begin{array}{l} \nu o_1.\nu o_2.([\![v_f]\!]_{o_1} \mid [\![v_a]\!]_{o_2} \mid \\ \quad \nu h_r.\nu write.\nu done.o_1(f_m).o_2(h).h(read, len).(\\ \qquad \mathsf{Array}(h_r, write, len, done) \mid \\ \qquad \nu x.(\overline{read}\langle x\rangle \mid \mathsf{Map}(x, f_m, write)) \mid \\ \qquad done().\overline{o}\langle h_r\rangle \\ )) \end{array} \tag{4.9}$$

In $[\![e]\!]_o$ we can take the reductions inside $[\![v_a]\!]_{o_2}$, then send on o_1 and o_2, and communicate on h and *read*. All these reductions are administrative, due to the v_f and v_a being values.

$$\begin{array}{rl} [\![e]\!]_o \overset{\circ}{\Rightarrow} P = & \nu h_r.\nu write.\nu done.\nu a_o.\ldots.\nu a_{n-1}.(\\ & \quad S_0 \mid \cdots \mid S_{n-1} \mid \\ & \quad !f_m(x, r).[\![e_b]\!]_r \mid \\ & \quad \mathsf{Array}(h_r, write, n, done) \mid \\ & \quad \nu x.(\mathsf{Map}(x, f_m, write) \mid \overline{x}\langle a_0\rangle \mid \cdots \mid \overline{x}\langle a_{n-1}\rangle) \mid \\ & \quad done().\overline{o}\langle h_r\rangle \\ &) \end{array} \tag{4.10}$$

Here, S_i denotes the support information for the handle a_i after $[\![v_i]\!]_{o_i}$ sends on o_i, i.e. $[\![v_i]\!]_{o_i} \mid o_i(x).W \overset{\circ}{\Rightarrow} \nu a_i.(S_i \mid W\{{}^{a_j}/_{x}\})$. If v_i is a number then a_i is the same number and $S_i = \mathbf{0}$, and we ignore the restriction on a_i. Otherwise, if v_i is a tuple, array, or function, then a_i is a handle and S_i is the supporting important for the value. We know that $[\![e]\!]_o$ always reaches P through $\overset{\circ}{\Rightarrow}$, and that any intermediate process between $[\![e]\!]_o$ and P is closed. Therefore, by lemma WABB 6 $[\![e]\!]_o \approx_a P$.

We now consider the translation $[\![e']\!]_o$, where $e = [v_f\ v_0, \ldots, v_f\ v_{n-1}]$.

$$
\begin{aligned}
[\![e']\!]_o = {} & \nu handle.\nu write.\nu done.(\mathsf{Array}(handle, write, n, done) \mid \\
& \quad O_0 \mid \cdots \mid O_{n-1} \mid \\
& \quad done().\bar{o}\langle handle\rangle \\
&) \text{ where} \\
& \quad O_i = \nu o_i.([\![v_f\ v_i]\!]_{o_i} \mid o_i(w_i).\overline{write}\langle i, w_i\rangle) \\
& \qquad = \nu o_i.(\nu o_a.\nu o_b.(\\
& \qquad\qquad \nu f_m.(!f_m(x,r).[\![e_b]\!]_r \mid \overline{o_a}\langle f_m\rangle) \mid \\
& \qquad\qquad \nu a_i.(S_i \mid \overline{o_b}\langle a_i\rangle) \mid \\
& \qquad\qquad o_a(f).o_b(x).\bullet\overline{f}\langle x, o_i\rangle \\
& \qquad\quad) \mid o_i(w_i).\overline{write}\langle i, w_i\rangle) \\
& \qquad \approx_a Q = \nu o_i.(\nu f_m.\nu a_i.(\\
& \qquad\qquad\quad !f_m(x,r).[\![e_b]\!]_r \mid S_i \mid \\
& \qquad\qquad\quad \bullet\overline{f_m}\langle a_i, o_i\rangle \\
& \qquad\qquad) \mid o_i(w_i).\overline{write}\langle i, w_i\rangle)
\end{aligned}
$$

Here, each O_i can take administrative reductions by sending on o_a, and o_b, i.e. $O_i \overset{\circ}{\rightarrow} O_i^{\omega} \overset{\circ}{\rightarrow} O_i'$, where O_i^{ω} is an intermediate process. In that both O_i and O_i^{ω} are closed (*write* is unrestricted but is guarded) we know by lemma WABB 6 that $O_i \approx_a O_i'$. Also by lemma OK 2 it can be shown that $!f_m(x,r).[\![e_b]\!]_r$ and S_i can be moved out along with its restriction on a_i. Applying these two observations yields equation (4.11) with processes O_i'' which is O_i' where S_i, νa_i, and $!f_m(.)[\![e_b]\!]_r$ have been moved out from O_i''.

$$
\begin{aligned}
[\![e']\!]_o \approx_a {} & \nu handle.\nu write.\nu done.\nu f_m.\nu a_0.\ldots.\nu a_{n-1}.(\\
& \quad \mathsf{Array}(handle, write, n, done) \mid \\
& \quad O_0'' \mid \cdots \mid O_{n-1}'' \mid \\
& \quad S_0 \mid \cdots \mid S_{n-1} \mid \\
& \quad !f_m(x,r).[\![e_b]\!]_r \mid \\
& \quad done().\bar{o}\langle handle\rangle \\
&) \text{ where} \\
& \quad O_i'' = \nu o_i.(\bullet\overline{f_m}\langle a_i, o_i\rangle \mid o_i(w_i).\overline{write}\langle i, w_i\rangle)
\end{aligned}
\tag{4.11}
$$

Now we want to show that P in equation (4.10) is weak administrative barbed bisimilar with equation (4.11). The only difference being that P has $\nu x.(\mathsf{Map}(x, f_m, write)\ldots)$ instead of $O_0'' \mid \ldots$. Thus, we claim that these two are weak administrative barbed bisimilar related, in equation (4.12).

$$O_0'' \mid \cdots \mid O_{n-1}'' \approx_a \nu x.(\mathsf{Map}(x, f_m, write) \mid \overline{x}\langle a_0\rangle \mid \cdots \mid \overline{x}\langle a_{n-1}\rangle) \tag{4.12}$$

We propose the witness R, and show that R^s is a weak administrative barbed bisimulation. Here, $O_i^2 = Q_i''$, $O_i^3 = o_i(w_i).\overline{write}\langle i, w_i\rangle$ is O_i^2 after sending on f_m, and lastly $O_i^4 = \overline{write}\langle i, M\rangle$ is O_i^3 after sending on o_i.

$$R = \{(C[\prod_{i\in I} O_i^{k_i} \mid \prod_{j\in J} O_i''],\quad C[\prod_{i\in I} O_i^{k_i} \mid \nu x.(\mathsf{Map}(x, f_m, \mathit{write}) \mid \prod_{j\in J} \overline{x}\langle j, a_j\rangle)])$$
$$\mid C \in \mathcal{C}_s, J\cup I = \{0,\dots,n-1\}, k_i \in \{2,3,4\}\}$$

We consider pairs $(C[X], C[Y]), (C[Y], C[X]) \in R^s$, and $X = \prod_{i\in I} O_i^3 \mid \prod_{i\in J} O_i''$ and $Y = \prod_{i\in I} O_i^3 \mid \nu x.(\mathsf{Map}(x, f_m, \mathit{write}) \mid \prod_{j\in J} \overline{x}\langle j, a_j\rangle)$, for the sets $I\cup J = \{0,\dots,n-1\}$. Then if $C[Y] \overset{\circ}{\rightarrow}{}^* W$ for some W, we know that either some $O_i^{k_i}$ has communicated with C or Map communicated with a $\overline{x}\langle u, a_u\rangle$. In the former case, $C[X]$ can follow because X has $O_u^{k_u}$ as a sub-process. In the latter case, Map spawns a process $\nu r.(\bullet\overline{f_m}\langle a_u, r\rangle \mid r(\mathit{value}').\overline{write}\langle j, \mathit{value}'\rangle$, which is structurally congruent with O_u'' through alpha conversion. Then $C[X]$ can follow with zero reductions and $(C[X], C[\prod_{i\in I} O_i^{k_i} \mid O_u'' \mid \nu x.(\mathsf{Map}(\dots) \mid \prod_{j\in J\setminus u} \overline{x}\langle j, a_j\rangle)]) \in R^s$. The same argument can be repeated for if $C[X] \xrightarrow{s} W$ for some other W.

Using equation (4.12) in $[\![e']\!]_o$, we can state that $[\![e']\!]_o \approx_a P \approx_a [\![e]\!]_o$, and by lemma WABB 4 $[\![e]\!]_o \approx_a [\![e']\!]_o$.

Iota $\boxed{e = \mathit{size}\{v\}}$ This case can be shown similarly to size and map.

Propagation Considering the rewrites in figure A.7, where sub-expressions of indexing, application, array, tuple, let, and the if construct, are rewritten. Here, some sub-expression e_i in e is rewritten to e_i'. By the translation of e, we know that e_i is unguarded. $e_i \rightharpoondown e_i'$ might itself happen by a propagation rule, but similarly to the proof of lemma OK 1, this can be assumed to hold for e_i. Therefore, $[\![e_i]\!]_{o_i} \approx_a [\![e_i']\!]_{o_i}$ and $[\![e]\!]_o \approx_a [\![e']\!]_o$. □

Lemma OK 3 Let R be the relation $R = \{(e, [\![e]\!]_o) \mid e \in \mathcal{B}\}$, then R is an operational correspondence.

Proof Here, we must show that for every pair $(e, P) \in R$ the rules for operational correspondence are fulfilled. Due to the construction of R we know that $P = [\![e]\!]_o$. We only consider pairs where $e \rightarrow e'$ and where $[\![e]\!]_o$ contains $\bullet$. By extension of this, we are not considering values.

Array $\boxed{e = [e_1, \dots, e_n]}$ Let us first consider that $e \rightarrow e'$, and from the ButF semantics, we know that there must exist an i such that $e_i \rightarrow e_i'$. Therefore we show that $[\![e]\!]_o \overset{\circ}{\Rightarrow}\overset{\bullet}{\rightarrow} p$ such that $P \approx_a [\![e']\!]_o$, in that $(e', [\![e']\!]_o) \in r$. Here, $[\![e]\!]_o$ contains $\nu o_i.([\![e_i]\!]_{o_i} \mid o_i(v_i).\overline{write}\langle i, v_i\rangle)$. We assume that $(e_i, [\![e_i]\!]_o) \in r$, and therefore know that $[\![e_i]\!]_{o_i} \overset{\circ}{\Rightarrow}\overset{\bullet}{\rightarrow} Q$ such that $Q \approx_a [\![e_i']\!]_{o_i}$. Let P be process $[\![e]\!]_o$ with the sub-process $[\![e_i]\!]_{o_i}$ replaced by Q. In that $[\![e_i]\!]_{o_i}$ is unguarded in $[\![e]\!]_o$, we know that $[\![e]\!]_o \overset{\circ}{\Rightarrow}\overset{\bullet}{\rightarrow} P$, and that $P \approx_a [\![e']\!]_o$.

Vice versa, we show that if $[\![e]\!]_o \overset{\circ}{\Rightarrow}\overset{\bullet}{\rightarrow} P$, then $P \dot{\approx}_a [\![e']\!]_o$ where $e \rightarrow e'$. Given that the translation of array does not have $\bullet$, the important reduction must happen inside $[\![e_i]\!]_{o_i}$. The translation ensures that $[\![e_i]\!]_{o_i}$ can only be observed on o_i, and therefore the different $[\![e_i]\!]_{o_i}$ cannot reduce together. We know there exists a j, such that the important reduction happens in $[\![e_j]\!]_{o_j}$, and because $(e_j, [\![e_j]\!]_o) \in R$, we know that $[\![e_j]\!]_{o_j} \overset{\circ}{\Rightarrow}\overset{\bullet}{\rightarrow} Q$ for some Q and e'_j where $Q \dot{\approx}_a [\![e'_j]\!]_{o_j}$ and $e_j \rightarrow e'_j$. We can then select e' as e where e_j has been replaced by e'_j, and then $P \dot{\approx}_a [\![e']\!]_o$. This is because P and $[\![e']\!]_o$ only differ by administrative reduction (for example in some other $[\![e_i]\!]_{o_i}$ for $i \neq j$).

Tuple $\boxed{e = (e_1, \ldots, e_n)}$ Follows from the same argument as **Array**.

Indexing $\boxed{e = e_1[e_2]}$ Operational correspondence requires that if $e \rightarrow e'$ then $[\![e]\!]_o$ $\overset{\circ}{\Rightarrow}\overset{\bullet}{\rightarrow} P$ such that $P \dot{\approx}_a [\![e']\!]$. The translation of indexing is defined as seen below.

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

There are three rules for indexing in ButF (E-Index, E-Index-1, and E-Index-2). On the other hand, in Eπ, there is the translation for the array ($[\![e_1]\!]_{o_1}$) and the expression to define the desired index ($[\![e_2]\!]_{o_2}$). The E-Index-1/2 rules are used to evaluate sub-expressions e_1 and e_2. Because $(e_1, [\![e_1]\!]_0) \in R$, if $e_1[e_2] \rightarrow e'_1[e_2]$ then $[\![e_1]\!]_{o_1} \overset{\circ}{\Rightarrow}\overset{\bullet}{\rightarrow} \dot{\approx}_a [\![e'_1]\!]_{o_1}$. Then because $[\![e_1]\!]_{o_1}$ is unguarded in $[\![e]\!]_o$, proof 10 holds.

$$[\![e]\!]_o \overset{\circ}{\Rightarrow}\overset{\bullet}{\rightarrow} \dot{\approx}_a \nu o_1.\nu o_2.([\![e'_1]\!]_{o_1} \mid [\![e_2]\!]_{o_2} \mid o_1(h).o_2(i).[i \geq 0] \bullet h \cdot i(v).\bar{o}\langle v\rangle, \mathbf{0}) = [\![e']\!]_o$$

The same has to hold for e_2. These have to be assumed to hold if all other cases are operationally correspondent since e_1 and e_2 are in R.

The actual indexing operation (E-Index) is what is of relevance here. Here, we know that if $v_1[v_2] \rightarrow v_3$ then we have to have the corresponding operation $[\![v_1[v_2]]\!]_o \overset{\bullet}{\Rightarrow} \dot{\approx}_a [\![v_3]\!]_o$. Because $e \rightarrow e'$ by E-Index, we know that e_1 is an array of length m and e_2 is an integer less than m. With the translation $\nu o_1.\nu o_2.([\![e_1]\!]_{o_1} \mid [\![e_2]\!]_{o_2} \mid o_1(h).o_2(i).[i \geq 0] \bullet h \cdot i(v).\bar{o}\langle v\rangle$) we know that they are ready to send on their o after some administrative reductions channels by lemma OK 1. The translation thus proceeds to send the handle of the array via o_1 and the value is sent on o_2. These are administrations reduction and are thus covered by the $\overset{\circ}{\Rightarrow}$ reductions.

This reduces the program down to $\nu h.(Q_h \mid \bullet h \cdot i(v).\bar{o}\langle v\rangle)$, where Q_h is the leftovers from the array $[\![e_1]\!]_{o_1}$ and i is the index from e_2. Next is the composite name together with $\bullet$, which is defined as an important reduction, and is expressed by the $\overset{\bullet}{\rightarrow}$ arrow: $\ldots \overset{\bullet}{\rightarrow} \nu o_1.\nu o_2.(Q_h \mid \bar{o}\langle v'\rangle)$. Lastly, the value is received internally as v' and returned along the out-channel (o). The still existing array Q_h can now be garbage collected by lemma WABB 3.

We must also show that if $[\![e]\!]_o \overset{\circ}{\Rightarrow}\overset{\bullet}{\rightarrow} P$ then we can find e' such that $P \dot{\approx}_a [\![e']\!]_o$ and $e \rightarrow e'$. The important reduction can either happen inside either $[\![e_1]\!]_{o_1}$ or $[\![e_2]\!]_{o_2}$

(very similar to array), or when receiving from the composite name $h \cdot i$. In the first case, we can find a e' much like in array and function application. In the latter case, we know that e_2 and i are integers that are greater or equal to zero and that some process is listening on $h \cdot i$. This is only the case if e_2 is an array of size larger than i. With this, we know that $e \rightarrow$ by E-INDEX.

Application $\boxed{e \coloneqq e_1\ e_2}$ There are two cases for which $e \rightarrow e'$. One case is when sub-expressions e_1 or e_2 can reduce. In that $[\![e_1]\!]_{o_1}$ and $[\![e_2]\!]_{o_2}$ appear unguarded in $[\![e]\!]_o$ and since $\{(e_1, [\![e_1]\!]_o), (e_2, [\![e_2]\!]_o)\} \subseteq R$, we know that $[\![e]\!]_o$ can match $\overset{\circ}{\Rightarrow}\overset{\bullet}{\rightarrow} \dot{\approx}_a$.

The second case is when $e_1 \not\rightarrow \wedge e_2 \not\rightarrow$. Here, E-BETA can take an important reduction. These are matched by the translation of application.

$$
\begin{array}{lr}
\nu o_1.\nu o_2.([\![e_1]\!]_{o_1} \mid [\![e_2]\!]_{o_2} \mid o_1(f).o_2(x).\bullet\overline{f}\langle x,o\rangle) & \overset{\circ}{\Rightarrow} \\
\nu o_1.\nu o_2.(\nu f'.\overline{o_1}\langle f'\rangle.(!f'(x,r).[\![e_b]\!]_r) \mid \nu v.(\overline{o_2}\langle v\rangle \mid S) \mid o_1(f).o_2(x).\bullet\overline{f}\langle x,o\rangle) & \overset{\circ}{\Rightarrow} \\
\nu v.\nu f'.(!f'(x,r).[\![e_b]\!]_r \mid \bullet\overline{f'}\langle v,o\rangle) \mid S & \overset{\bullet}{\rightarrow} \\
\nu f'.(!f'(x,r).[\![e_b]\!]_r) \mid \nu v.(F_o \mid S) & \dot{\approx}_a \\
\nu v.(F_o \mid S) &
\end{array}
$$

First, the expressions are evaluated to values such that they are ready to send on the out-channels. This results in a guarded replicated function server for e_1 and a value ready to be sent for e_2. Afterward, the administrative reductions, in the form of communicating along the out-channels, are performed.

We know that e_1 is an abstraction, $\lambda x.e_b$, and therefore $[\![e_1]\!]_{o_1} = \nu f.(!f(x,r) .[\![e_b]\!]_r \mid \overline{o_1}\langle f\rangle)$. Also note that S is the process needed to maintain value v, i.e. $[\![e_2]\!]_{o_2} \dot{\approx}_a \nu a.(S \mid \overline{o_2}\langle v\rangle)$ such that S is only observable on a or $\overline{a}$.

After the two sub-processes have sent their value on o, we can send on f' which is marked by a $\bullet$. By sending (v,o) an instance of $[\![e_b]\!]_r$ is unguarded, where the name of the return channel is substituted with the name of the out-channel (o) together with the value (v).

We let F_o denote the function body $[\![e_b]\!]_r$ with return channel o and the value from $[\![e_2]\!]_{o_2}$, i.e. $F_o = [\![e_b]\!]_o\{v/x\}$. F_o then corresponds to the translation of $e' = e_b\{x \coloneqq e_1\}$ by conjecture 2, and thus $[\![e]\!]_o \overset{\circ}{\Rightarrow}\overset{\bullet}{\rightarrow} \dot{\approx}_a [\![e']\!]_o$.

In the above, we assume that $e_1 = \lambda x.e_b$, but e_1 can also be a tuple abstraction, i.e. $e_1 = \lambda(x_0,\ldots,x_n).e_b$. Because $e \rightarrow e'$, we know that $e_2 = (v_1,\ldots,v_n)$, and can follow the same argument as above.

If $[\![e]\!]_o \overset{\circ}{\Rightarrow}\overset{\bullet}{\rightarrow} P$ then we must show that e' exists such that $P \dot{\approx}_a [\![e']\!]_o$ and $e \rightarrow e'$. Like with arrays, if $\overset{\bullet}{\rightarrow}$ happens entirely inside either $[\![e_1]\!]_{o_1}$ or $[\![e_2]\!]_{o_2}$ then, we can select $e' = e_1'\ e_2$ or $e' = e_1\ e_2'$. If $\overset{\bullet}{\rightarrow}$ happens when sending on f, then both $[\![e_1]\!]_{o_1}$ and $[\![e_2]\!]_{o_2}$ can send on o after some administrative reductions. Therefore by conjecture 1 e_1 and e_2 must be values. Also $[\![e_1]\!]_{o_1}$ must send the name of a function channel on o_1 and therefore we know that $e_1 = \lambda x.e_b$ or $e_1 = \lambda p.e_b$. Therefore we can take E-Beta from e to $e' = e_b\{p := e_2\}$.

Name binding $\boxed{e := \textbf{let } x = e_1 \textbf{ in } e_2}$ If $e \to e'$ then $[\![e]\!]_o$ must follow with $\overset{\circ}{\Rightarrow}\overset{\bullet}{\to} \approx_a$. There are two rules for name binding in ButF, E-Let and E-Let-1. E-Let-1 is for reducing e_1, and E-Let handles the case where e_1 is a value. Let us first consider when e_1 is not a value. Assuming that $e_1 \to e_1'$, we use E-Let-1 and thus $e' = \textbf{let } x = e_1' \textbf{ in } e_2$, where $e \to e'$. Because $(e_1, [\![e_1]\!]_o) \in R$ and $[\![e_1]\!]_{o_1}$ is unguarded in $[\![e]\!]_o$, $[\![e]\!]_o$ can follow with $\overset{\circ}{\Rightarrow}\overset{\bullet}{\to} P$ such that $P \approx_a [\![e']\!]_o$.

The other case is if e_1 is a value, and there exists e' where $e \to e'$ through E-Let. $[\![e]\!]_o$ should then follow with $\overset{\circ}{\Rightarrow}\overset{\bullet}{\to}$ to P, such that $P \approx_a [\![e']\!]_o$. The translation of e can be seen below.

$$[\![e]\!]_o = \nu o_1.([\![e_1]\!]_{o_1} \mid \bullet o_1(x).[\![e_2]\!]_o)$$

We can then take any administrative reduction in $[\![e_1]\!]_o$ followed by the important reduction by sending on o_1, yielding P. We can then use conjecture 2 to state $P \approx_a [\![e_2\{x := e_1\}]\!]_o$. From E-Let we know that $e' = e_2\{x := e_1\}$ and have therefore shown that $[\![e]\!]_o \overset{\circ}{\Rightarrow}\overset{\bullet}{\to} \approx_a [\![e']\!]_o$. Like with application, the same argument can be repeated if $e = \textbf{let } p = e_1 \textbf{ in } e_2$.

If $[\![e]\!]_o \overset{\circ}{\Rightarrow}\overset{\bullet}{\to} P$, then there must exists e' such that $e \to e'$ and $[\![e']\!]_o \approx_a P$. P is one of two cases: either $\overset{\circ}{\Rightarrow}\overset{\bullet}{\to}$ happened entirely inside $[\![e_1]\!]_{o_1}$ or $\overset{\circ}{\Rightarrow}\overset{\bullet}{\to}$ is the single communication on o_1. In the first case $e' = \textbf{let } e_1' \textbf{ in } e_2$ where $e_1 \to e_1'$, and thus because $(e_1, [\![e_1]\!]_o) \in R$ then we know $P \approx_a [\![e']\!]$. In the second case, we know that e_1 is a value from conjecture 1 in that $[\![e_1]\!]_{o_1}$ can communicate on o_1. Therefore $e' = e_2[x := e_1]$ and by previous argument $P \approx_a [\![e']\!]_o$.

Conditional $\boxed{e = \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3}$

There are three rules for the if/else conditionals in ButF, E-If-1 which allows e_1 to reduce to a value, E-If-True if the conditional is non-zero, and E-If-False if the condition is zero. The translation for e is as seen below.

$$\nu o_1.([\![e_1]\!]_{o_1} \mid o_1(v).[v \neq 0]\ [\![e_2]\!]_o, [\![e_3]\!]_o)$$

For the first case E-If-1, we know that any reduction done by e_1, can be done by $[\![e_1]\!]_{o_1}$ since $[\![e_1]\!]_{o_1}$ is unguarded and $(e_1, [\![e_1]\!]_o) \in R$. Once e_1 is done and can send some term (M) on o_1, there is only one reduction left. This reduction reduces $[M \neq 0]\ [\![e_2]\!]_o, [\![e_3]\!]_o$ to either $[\![e_2]\!]_o$ or $[\![e_3]\!]_o$. Since $e \to$ and $e_1 \not\to$, expression e_1 must be a value, and thus either E-If-True is matched and $[M \neq 0]\ [\![e_2]\!]_o, [\![e_3]\!]_o \overset{\bullet}{\to} [\![e_2]\!]_o$ or E-If-False is matched and $[M \neq 0]\ [\![e_2]\!]_o, [\![e_3]\!]_o \overset{\bullet}{\to} [\![e_3]\!]_o$.

In the other case when $[\![e]\!]_o \overset{\circ}{\Rightarrow}\overset{\bullet}{\to} P$, we can show that e' exists such that $P \approx_a e'$ and $e \to e'$, in much the same way as with name binding. □

With all cases covered and shown to satisfy the operational correspondence, the translation as a whole is shown to have the operational correspondence property. We denote $e \gtrless_{ok} [\![e]\!]_o$ when $(e, [\![e]\!]_o) \in R$.

Theorem 2 For any e it holds that $e \gtrless_{ok} [\![e]\!]_o$ and if there exists $e' \in \mathcal{V}$, such that $e \to^* e'$ it holds that $[\![e]\!]_o \overset{\bullet}{\Rightarrow}^* \approx_a [\![e']\!]_o$.

PROOF By lemma OK 3, we know that a witness exists such that $e \dot{\approx}_a [\![e]\!]_o$. Therefore if $e \to^* e'$ then $[\![e]\!]_o \overset{\circ}{\Rightarrow} \overset{\bullet}{\to} \dot{\approx}_a \overset{\circ}{\Rightarrow} \overset{\bullet}{\to} \dot{\approx}_a \ldots \overset{\circ}{\Rightarrow} \overset{\bullet}{\to} \dot{\approx}_a [\![e']\!]_o$. By repeated usage of lemma WABB 5 each $\approx_a$ can be moved to the end such that $e_o \overset{\bullet}{\Rightarrow}^*$ and the theorem holds.□

Chapter 5

Conclusion

By analyzing the two paradigms in regards to translating them, the two languages, Eπ and BUTF were created. Their syntaxes and semantics were defined to be able to reason about them. With this foundation, a generic translation was proposed. By then creating a framework to further reason about the translations' behavior, in the form of iterating the requirements for weak bisimulations, it enabled us to define and prove the correctness. We, therefore, discuss this process and our findings, some of which is followed by interesting ideas some of which were deprioritized due to time constraints.

Changes over prior work This report features several improvements over our previous work [9]. The first improvement is the addition of a small-step semantics, so BUTF's reductions are now able to be described in more detail. The syntax for BUTF was also subject to some changes. The **prog** and **def** syntactic elements were completely removed and **iota** is introduced as a built-in function. Furthermore, support for tuples and patterns has been added, allowing the language to be more similar to FUTHARK. Lastly, recursion is added to allow for easier implementation of derived primitive candidates. While this is a feature FUTHARK does not support, adding recursion does not extend the expressivity of BUTF beyond the expressivity of FUTHARK.

We also change the Eπ (formerly ButF-π) with minor changes. Some of those being the removal of bounded replication as well as newly labeled semantics to ensure that broadcasting casts to all receivers.

5.1 Contribution and Results

This report started with a focus on the analysis of parallelism in regard to Futhark. Here, the work and the span complexities were described as fundamental metrics for the benefit of using concurrent approaches. By analyzing FUTHARK, the aspects deemed interesting and important were isolated and condensed into the smaller language BUTF. Its smaller syntax combined with a small-step semantics defines the foundation of further analysis.

We introduce the π-calculus as the target language due to its different kind of properties such as natural concurrency. The π-calculus was extended to better facilitate a translation which we then called Eπ. We define equivalence on Eπ processes through the notion of observability, contexts, and bisimulation.

With these two languages, a generic translation has been created such that every BUTF program is able to be translated into Eπ. To reduce the complexity of the translation, several BUTF expressions were implemented in the language itself via other BUTF expressions, thus simplifying the language. These implementations were then analyzed in their complexity and dependencies. Afterward, possible subsets of these derived expressions were considered, in which one of them was chosen to reduce the complexity of the proof while maintaining the asymptotic complexity of work and span.

To show the correctness of the translation, we introduce the notion of operational correspondence as a relation. Here, the translation is modified with annotations on important and administrative reductions. Through a case-by-case analysis, the translation is proven to be correct in regard to this operational correspondence. This resulted in proposing a correct translation for core concepts of FUTHARK into an extended π-calculus, that can be used for further analysis.

5.2 Basic Un-Typed Futhark and Extended π-Calculus

BUTF is a language that reduces the extensive nature of FUTHARK by only keeping the features deemed relevant. On the other hand, Eπ is an extension of the π-calculus to facilitate said relevant features in a more natural way. These changes to the languages come with advantages and disadvantages.

Type systems Since BUTF is un-typed it made it easier to implement the language. However, this comes at the disadvantage of not being able to clearly and unambiguously validate the correctness of programs. For example, indexing an abstraction is valid syntax, but results in undefined behavior. Type systems, on both the BUTF side as well as the Eπ side of the translation, could also have helped the translation itself by defining a stricter correspondence via types.

Omitted Futhark features Futhark comes with various optimization strategies compared to BUTF, some of which have been left out despite being beneficial for concurrency optimizations.

In-place updating of arrays allows the compiler to directly update an array instead of having to construct an updated copy. Here, the programmer can tell the compiler that the argument has no other references that could be affected when updating. This is an especially efficient optimization with sequences of memory-intensive operations where the intermediate results are not used. While this optimization is relevant and useful for real hardware, it can limit parallelism and is therefore omitted from BUTF. [21]

The Futhark compiler also has fusion rules for their SOACs that help the compiler to match patterns of SOACs and restructure or fuse them into a more optimized structure that allows for chunking the workload. In these rewriting rules, the chunk size is dynamic which makes it easier to utilize the hardware at hand, but the parallelization can be maximized by setting the chunk size to one. [21]

This report did not further look into incorporating these features since the hardware aspect was abstracted away from by using the π-calculus as process model which is unbounded.

Expressivity of Eπ The inclusion of broadcasting, composite names, and using the applied π-calculus are deemed good choices to reduce the complexity of the translation. Broadcasting has been shown to be more expressive than π-calculus by Ene and Muntean [34], and similarly Carbone and Maffeis [18], however how these extensions relate to each other has not been explored. Due to this, composite names might be expressible by the broadcast π-calculus or vice versa, and the expressivity of Eπ is therefore unknown.

Eπ as a computational model Since Eπ is an extended π-calculus, additional expressivity has been added. However, the origin of this project lies in Futhark which is more closely tied to computers and their instructions as we know them. Broadcast is an extension for the π-calculus that allows the propagation of values to an unbounded amount of processes. This type of single reduction is not nearly as easy to represent in computer-executable languages. This means that using broadcast is, despite providing interesting and useful functionality, not easily representable for a complexity analysis to optimize the execution of computer code. This is because it would not be fair to assume that one action, broadcast, can ever be matched by limited computer resources.

The composite names extension enabled a natural encoding of arrays, which is a data structure that is closely related to the sequential memory of computers. While memory can be indexed with pointers and simple arithmetic (to a degree), the same applies to composite names, which makes this extension a reasonable inclusion.

These thoughts need to be considered when designing a proper and representable complexity analysis.

5.3 Translation

This section discusses the translation, being the main contribution of this report.

Choice of primitives By analyzing the possible sets of primitive candidates, based on our derived implementations, the ideal set for our use case was determined. However, there might exist better methods of implementing these, which might further ease translation and provide better asymptotic complexity. There were different possible sets with various qualities, such as being the smallest, the lowest span complexity, or the easiest to prove. The choice that was made focused on the ease of proving whilst still aiming to not increase the asymptotic span complexity. Since the set found and chosen is considerably smaller, simpler to prove, and even preserves the asymptotic work and span complexities of all primitive candidates, it helped the further analysis considerably.

Important reductions and bisimulation When proving the correctness of the translation, we introduce the notion of important reductions through the $\bullet$ syntax. These present a "blocker" which can only be removed with an important reduction $\overset{\bullet}{\rightarrow}$. This block prevents the translated program from executing ahead of the source program. This is useful in the operational correspondence, in that reduction of e and $[\![e]\!]_o$ both meet at e' and $[\![e']\!]_o$ when $e \rightarrow e'$. Without this blocker (and important reductions) one can still propose a relaxed operational correspondence, where $[\![e]\!]_o$ can skip over $[\![e']\!]_o$, but must be able to reach some later $[\![e'']\!]_o$ where $e \rightarrow^* e''$. However, it was anticipated that such a correspondence was harder to prove.

The formulation of two operational correspondences During the report two definitions of operational correspondence are proposed. The first is an initial approach inspired by Sangiorgi [16]. Our approach differs in that the translation is allowed to take multiple steps to match each reduction in BUTF.

However, we feared that this definition would be cumbersome to prove, in that we would need to consider the many different translation rules of chapter 3. Instead, we formulated the administrative operational correspondence which related expressions and processes mush like bisimulation, inspired by Milner, Amadio, Thomsen, and Thomsen [15, 17]. Then, as a proof by co-induction, we propose a relation containing any pair of expressions and their translation and show that this is a operational correspondence. A proof therefore only needed to consider each pair in isolation.

However, we found that showing the validity of these pairs was still rather cumbersome, and it is unclear whether a proof via co-induction was simpler. Maybe a distinction between soundness and completeness would have been beneficial as these could then have been proven separately.

Translation of values BUTF has several different types of values: number, array, tuple, and abstraction. Numbers are easily represented in Eπ, but arrays, tuples, and abstractions are instead represented by a handle to some replicated resource, functioning much like a pointer to some memory.

This handle-representation of values in the translation has one major oversight. The semantics of BUTF at run-time is able to distinguish between the different value types, thus only allowing reductions when the expression has no errors. The translation does not tag run-time values with their type, meaning that it cannot distinguish between an array handle and a tuple handle. This is solved by requiring different interactions with each value type: calling a function ($\overline{f}\langle v, r\rangle$), fetching array elements ($h \cdot i(v)$ or $h(\mathit{read}, \mathit{len})$), and fetching tuple elements ($h \cdot h(v_0, \ldots, v_{n-1})$). In the case of an error the translation stops, because either the communication arity or action on a handle does not match. While not an elegant nor robust solution, it does ensure that the translation stops reducing on an error. A more elegant and preferred solution is to implement sorts in Eπ, or have some run-time tagging alongside values.

Array translation This report features two different implementations of arrays. The main difference between them is the addition of the await process ensuring that an array cannot be used before all elements have been added to the array. Ensuring this is essential for the translation since it would otherwise make operational correspondence difficult. This is due to some elements of the array potentially being multiple reductions ahead compared to the rest of the array.

Additionally, several different approaches for the array implementation were considered. The earliest of these was implementing the index operator using a combination of broadcast and if statements on all elements of the array. This would allow for indexing without using composite names, but would also require more reductions every time indexing is used. Another possible addition discussed was cells for modifiable memory [9]. This idea is deemed unimportant for the translation, as FUTHARK's in-place updating does not add more concurrency. Thus to keep the translation as simple as possible, only read-only arrays were implemented.

Another change is removing broadcast from the Array when requesting array elements, in favor of sending the request to each element via replication, switching the channel on each request. The main reasoning behind this is to ensure that all elements of the array would receive all requests on the *read* channel and in case any elements arrive after the message has been sent, they would still receive it. One problem with this approach is, however, that it requires more overhead since it requires the creating of a new channel each time a message on *read* is received, as well as having to send the channel name to all elements, such that the communications do not interfere with each other. Furthermore, this also complicates the proof of operational correspondence and was thus deemed an impractical approach.

Parallelism and data-parallelism FUTHARK is a data-parallel language, in that array operations are efficiently mapped to the hardware at hand.

BUTF and Eπ are not parallel languages in themselves, however, one can design a cost model in which concurrent expressions and processes are run in parallel. Then, like FUTHARK, array operations become parallel, but also constructs like indexing and application have parallelizable aspects. This is because sub-expressions (e_1 and e_2) of BUTF application and indexing are reduced concurrently in both BUTF's semantics and the translation in Eπ. BUTF can therefore be parallel outside of the data-parallelism of FUTHARK.

5.4 Future Work

This project covered the translation, however, there are other interesting aspects that can be worked on in the future that did not make it into this report.

Cost model and complexity analysis The natural continuation of this work is to use the translation to analyze the parallel properties of the translated constructs. To do this a cost model is needed that needs to be designed to capture the work and span properties of processes. However, this has not been researched in this project due to time constraints which leaves us with only a translation that has been proven correct, but not useful. With such a cost analysis we would be able to compare the translation to FUTHARK to find potential weaknesses or speed-ups. Such a cost model could be implemented similarly to the $\bullet$ notation used to mark important reductions. In related work, a "tick notation" is used to analyze cost in a similar manner [35]. This would not only give insight into parallel properties of the constructs but also be a framework that can be extended and used to optimize the translation further.

Expressivity Analyzing the expressivity of Eπ is an interesting point of interest which could help discover how Eπ relates to similar π-calculi in regards to expressivity. This can help with the understanding of Eπ itself and also open up for related work on similar categories of π-calculi. Researching this could be interesting since it might open Eπ to already-created analysis methods.

We also believe that it is possible to create a correct translation without these extensions, however how this impacts the complexity of a translation is unknown and might be interesting to explore.

A transpiler implementation An actual transpiler would be an excellent base for further analysis while being able to automatically translate any program. This base could be extended in various ways such as testing, model checking, proving assistance, compiling, error handling, and more.

Running a π-calculus on hardware Since Eπ is a process calculus it could be interesting to research how to run it on hardware. While some research has already been done in regards to doing this with the π-calculus, there is none on our variant [36]. If one could either run Eπ directly on the hardware or on an abstract machine, it could be compared more directly to FUTHARK.

Bibliography

[1] Intel. *4th Generation Intel® Core™ i7 Processors.* URL: `https://ark.intel.com/content/www/us/en/ark/products/series/75023/4th-generation-intel-core-i7-processors.html#@Desktop` (visited on 05/17/2023).

[2] Intel. *13th Generation Intel® Core™ i9 Processors.* URL: `https://ark.intel.com/content/www/us/en/ark/products/series/230485/13th-generation-intel-core-i9-processors.html` (visited on 05/17/2023).

[3] Guy E. Blelloch. "Programming Parallel Algorithms". In: *Commun. ACM* 39.3 (1996), pp. 85–97. DOI: `10.1145/227234.227246`. URL: `https://doi.org/10.1145/227234.227246`.

[4] Khronos Group. *OpenCL API.* URL: `https://www.khronos.org/opencl/` (visited on 09/30/2022).

[5] nvidia. *CUDA Toolkit.* URL: `https://developer.nvidia.com/cuda-toolkit` (visited on 09/30/2022).

[6] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. "Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015.* Ed. by Kathleen Fisher and John H. Reppy. ACM, 2015, pp. 205–217. ISBN: 978-1-4503-3669-7. DOI: `10.1145/2784731.2784754`. URL: `https://doi.org/10.1145/2784731.2784754`.

[7] Troels Henriksen. "Design and Implementation of the Futhark Programming Language". English. PhD thesis. 2017.

[8] HyoukJoong Lee, Kevin J. Brown, Arvind K. Sujeeth, Tiark Rompf, and Kunle Olukotun. "Locality-Aware Mapping of Nested Parallel Patterns on GPUs". In: *47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2014, Cambridge, United Kingdom, December 13-17, 2014.* IEEE Computer Society, 2014, pp. 63–74. ISBN: 978-1-4799-6998-2. DOI: `10.1109/MICRO.2014.23`. URL: `https://doi.org/10.1109/MICRO.2014.23`.

[9] Lars Jensen, Chris Oliver Paulsen, and Julian Jørgensen Teule. “Constructs of the Futhark Programming Language Described in a pi-Calculus”. In: (Jan. 2023). URL: `https://projekter.aau.dk/projekter/da/studentthesis/constructs-of-the-futhark-programming-language-described-in-a-picalculus(3be42cfd-46d8-4308-81fd-4ff3863d97b3).html` (visited on 02/09/2023).

[10] Hans Hüttel. *Transitions and Trees - An Introduction to Structural Operational Semantics*. Cambridge University Press, 2010. ISBN: 978-0-521-14709-5. URL: `http://www.cambridge.org/de/academic/subjects/computer-science/programming-languages-and-applied-logic/transitions-and-trees-introduction-structural-operational-semantics`.

[11] Klaus Berkling. *Arrays and the Lambda Calculus*. Tech. rep. Syracuse University, 1990.

[12] Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2016. ISBN: 978-1316626221. URL: `https://www.cs.nott.ac.uk/~pszgmh/pih.html`.

[13] Troels Henriksen, Ken Friis Larsen, and Cosmin E Oancea. “Design and GPGPU performance of Futhark’s redomap construct”. In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. 2016, pp. 17–24.

[14] Martín Abadi, Bruno Blanchet, and Cédric Fournet. “The Applied Pi Calculus: Mobile Values, New Names, and Secure Communication”. In: *J. ACM* 65.1 (2018), 1:1–1:41. DOI: `10.1145/3127586`. URL: `https://doi.org/10.1145/3127586`.

[15] Robin Milner. “Functions as processes”. In: *Automata, Languages and Programming*. Vol. 443. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 167–180. ISBN: 9783540528265.

[16] Davide Sangiorgi. “From pi-Calculus to Higher-Order pi-Calculus - and Back”. In: *TAPSOFT’93: Theory and Practice of Software Development, International Joint Conference CAAP/FASE, Orsay, France, April 13-17, 1993, Proceedings*. Ed. by Marie-Claude Gaudel and Jean-Pierre Jouannaud. Vol. 668. Lecture Notes in Computer Science. Springer, 1993, pp. 151–166. ISBN: 3-540-56610-4. DOI: `10.1007/3-540-56610-4\_62`. URL: `https://doi.org/10.1007/3-540-56610-4%5C_62`.

[17] Roberto M. Amadio, Lone Leth Thomsen, and Bent Thomsen. “From a Concurrent Lambda-Calculus to the Pi-Calculus”. In: *Fundamentals of Computation Theory, 10th International Symposium, FCT ’95, Dresden, Germany, August 22-25, 1995, Proceedings*. Ed. by Horst Reichel. Vol. 965. Lecture Notes in Computer Science. Springer, 1995, pp. 106–115. ISBN: 3-540-60249-6. DOI:

10.1007/3-540-60249-6_43. URL: https://doi.org/10.1007/3-540-60249-6%5C_43.

[18] Marco Carbone and Sergio Maffeis. “On the Expressive Power of Polyadic Synchronisation in pi-calculus”. In: *9th International Workshop on Expressiveness in Concurrency, EXPRESS 2002, Satellite Workshop from CONCUR 2002, Brno, Czech Republic, August 19, 2002.* Ed. by Uwe Nestmann and Prakash Panangaden. Vol. 68. Electronic Notes in Theoretical Computer Science 2. Elsevier, 2002, pp. 15–32. DOI: 10.1016/S1571-0661(05)80361-5. URL: https://doi.org/10.1016/S1571-0661(05)80361-5.

[19] Hans Hüttel and Nuno Pratas. “Broadcast and aggregation in BBC”. In: *Proceedings Eighth International Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES 2015, London, UK, 18th April 2015.* Ed. by Simon Gay and Jade Alglave. Vol. 203. EPTCS. 2015, pp. 15–28. DOI: 10.4204/EPTCS.203.2. URL: https://doi.org/10.4204/EPTCS.203.2.

[20] DIKU. *7. Fusion and List Homomorphisms.* URL: https://futhark-book.readthedocs.io/en/latest/fusion.html (visited on 02/14/2023).

[21] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. “Futhark: purely functional GPU-programming with nested parallelism and in-place array updates”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017.* Ed. by Albert Cohen and Martin T. Vechev. ACM, 2017, pp. 556–571. DOI: 10.1145/3062341.3062354. URL: https://doi.org/10.1145/3062341.3062354.

[22] Robin Milner. “The Polyadic π-Calculus: a Tutorial”. In: *Logic and Algebra of Specification.* Ed. by Friedrich L. Bauer, Wilfried Brauer, and Helmut Schwichtenberg. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 203–246. ISBN: 978-3-642-58041-3.

[23] Robin Milner. *Communicating and mobile systems - the Pi-calculus.* Cambridge University Press, 1999. ISBN: 978-0-521-65869-0.

[24] Davide Sangiorgi and David Walker. *The Pi-Calculus - a theory of mobile processes.* Cambridge University Press, 2001. ISBN: 978-0-521-78177-0.

[25] “On Barbed Equivalences in pi-Calculus”. eng. In: *CONCUR 2001 - Concurrency Theory.* Vol. 2154. Lecture Notes in Computer Science. Germany: Springer Berlin / Heidelberg, 2001, pp. 292–304. ISBN: 9783540424970.

[26] Benjamin C. Pierce. *Types and programming languages.* MIT Press, 2002. ISBN: 978-0-262-16209-8.

[27] Futhark. *A Parallel Cost Model for Futhark Programs*. URL: `https://futhark-book.readthedocs.io/en/latest/parallel-cost-model.html` (visited on 04/04/2023).

[28] Richard E. Ladner and Michael J. Fischer. “Parallel Prefix Computation”. In: *J. ACM* 27.4 (1980), pp. 831–838. DOI: `10.1145/322217.322232`. URL: `https://doi.org/10.1145/322217.322232`.

[29] Guy E Blelloch. “Prefix sums and their applications”. In: (1990).

[30] Futhark. */prelude/array*. URL: `https://futhark-lang.org/docs/prelude/doc/prelude/soacs.html` (visited on 04/04/2023).

[31] DIKU. *Language Reference*. URL: `https://futhark.readthedocs.io/en/stable/language-reference.html` (visited on 10/13/2022).

[32] Futhark. */prelude/array*. URL: `https://futhark-lang.org/docs/prelude/doc/prelude/array.html` (visited on 04/04/2023).

[33] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. ISBN: 978-0-262-03384-8. URL: `http://mitpress.mit.edu/books/introduction-algorithms`.

[34] Cristian Ene and Traian Muntean. “Expressiveness of Point-to-Point versus Broadcast Communications”. In: *Fundamentals of Computation Theory, 12th International Symposium, FCT ’99, Iasi, Romania, August 30 - September 3, 1999, Proceedings*. 1999, pp. 258–268. DOI: `10.1007/3-540-48321-7\_21`. URL: `https://doi.org/10.1007/3-540-48321-7%5C_21`.

[35] Patrick Baillot and Alexis Ghyselen. “Types for Complexity of Parallel Computation in Pi-Calculus”. In: *Programming Languages and Systems*. Ed. by Nobuko Yoshida. Cham: Springer International Publishing, 2021, pp. 59–86. ISBN: 978-3-030-72019-3.

[36] André Seffrin and Sorin A Huss. “Hardware-accelerated execution of Pi-calculus reconfiguration schedules”. In: *2011 International Conference on Field- Programmable Technology*. IEEE. 2011, pp. 1–8.

Appendix A

Basic Un-Typed Futhark Small-Step Semantics

This is an overview of the entire small-step semantics for BUTF.

$$\frac{e \to e'}{[e_0, \ldots, e, \ldots e_n] \to [e_0, \ldots, e', \ldots, e_n]}\ [\text{E-ARR}] \qquad \frac{e_1 \to e_1'}{e_1[e_2] \to e_1'[e_2]}\ [\text{E-INDEX-1}]$$

$$\frac{e_2 \to e_2'}{v_1[e_2] \to v_1[e_2']}\ [\text{E-INDEX-2}] \qquad \frac{m \leq n \qquad n = |[\vec{v}\,]|}{[\vec{v}\,][m] \to v_m}\ [\text{E-INDEX}]$$

$$\frac{e \to e'}{(e_0 \ldots, e, \ldots, e_n) \to (e_0, \ldots, e', \ldots, e_n)}\ [\text{E-TUP}]$$

Figure A.1: Small-step semantics rules for arrays and tuples in BUTF.

$$\frac{}{(\lambda p \Rightarrow e)\ v \to e\{p := v\}}\ [\text{E-BETA}]$$

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2}\ [\text{E-APP-1}] \qquad \frac{e_2 \to e_2'}{e_1\ e_2 \to e_1\ e_2'}\ [\text{E-APP-2}]$$

Figure A.2: Application rules for BUTF, describing β-reduction and reduction inside application.

$$\frac{}{\textbf{let } p = v \textbf{ in } e \rightarrow e\{p := v\}}\ [\text{E-Let}]$$

$$\frac{e_1 \rightarrow e_1'}{\textbf{let } p = e_1 \textbf{ in } e_2 \rightarrow \textbf{let } p = e_1' \textbf{ in } e_2}\ [\text{E-Let-1}]$$

Figure A.3: The semantic rules for let bindings in ButF.

$$\frac{e_1 \rightarrow e_1'}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \rightarrow \textbf{if } e_1' \textbf{ then } e_2 \textbf{ else } e_3}\ [\text{E-If-1}]$$

$$\frac{v \neq 0}{\textbf{if } v \textbf{ then } e_2 \textbf{ else } e_3 \rightarrow e_2}\ [\text{E-If-True}] \qquad \frac{v = 0}{\textbf{if } v \textbf{ then } e_2 \textbf{ else } e_3 \rightarrow e_3}\ [\text{E-If-False}]$$

Figure A.4: The rules for conditionals in ButF.

$$\frac{}{\textbf{loop } p = e_1 \textbf{ for } x < e_2 \textbf{ do } e_3 \rightharpoondown [x = 0]\ \textbf{loop } p = e_1 \textbf{ for } x < e_2 \textbf{ do } e_3}\ [\text{E-Loop-Init}]$$

$$\frac{m < n}{\begin{array}{l}[x = m]\ \textbf{loop } p = v \textbf{ for } x < n \textbf{ do } e_3 \rightarrow \\ [x = \underline{m+1}]\ \textbf{loop } p = e_3\{p := v, x := m\} \textbf{ for } x < n \textbf{ do } e_3\end{array}}\ [\text{E-Loop-Iter}]$$

$$\frac{m \geq n}{[x = m]\ \textbf{loop } p = v \textbf{ for } x < n \textbf{ do } e_3 \rightarrow v}\ [\text{E-Loop-F}]$$

$$\frac{e_1 \rightarrow e_1'}{[x = m]\ \textbf{loop } p = e_1 \textbf{ for } x < e_2 \textbf{ do } e_3 \rightarrow \textbf{loop } p = e_1' \textbf{ for } x < e_2 \textbf{ do } e_3}\ [\text{E-Loop-1}]$$

$$\frac{e_2 \rightarrow e_2'}{[x = m]\ \textbf{loop } p = e_1 \textbf{ for } x < e_2 \textbf{ do } e_3 \rightarrow \textbf{loop } p = e_1 \textbf{ for } x < e_2' \textbf{ do } e_3}\ [\text{E-Loop-2}]$$

Figure A.5: The small-step semantics rules for the **loop** construct in ButF.

$$\frac{e_1 \rightharpoondown e_2 \quad e_2 \rightarrow e_2'}{e_1 \rightarrow e_2'} \text{[E-Rewrite]}$$

$$\frac{}{e_1 \,`e_2`\, e_3 \rightharpoondown e_2\ e_1\ e_3} \text{[E-Binop]} \qquad \frac{}{cf \rightharpoondown \underline{\mathrm{expand}(cf)}} \text{[E-Expand]}$$

Figure A.6: The rules for transforming the built-in functions in ButF.

$$\begin{aligned}
\odot\{v_1, v_2\} &\rightharpoondown \underline{v_1 \odot v_2} \\
\rhd\{v\} &\rightharpoondown \underline{\rhd v} \\
size\{v\} &\rightharpoondown \underline{|[\vec{v}]|} \\
concat\{v_1, v_2\} &\rightharpoondown [\vec{v_1}, \vec{v_2}] \\
iota\{v\} &\rightharpoondown [0, \ldots, \underline{v-1}] \\
map\{f, \vec{v}\} &\rightharpoondown [\overrightarrow{f\ v}] \\
reduce\{f, z, \vec{v}\} &\rightharpoondown z \,`f`\, v_0 \,`f`\cdots`f`\, v_n \\
scan\{f, z, \vec{v}\} &\rightharpoondown [a_0, \ldots, a_n] \\
&\qquad \text{where } a_i \rightharpoondown z \,`f`\, v_0 \,`f`\cdots`f`\, v_i
\end{aligned} \tag{A.1}$$

$$\begin{aligned}
\mathrm{expand}(\odot) &= \lambda x.\lambda y. \odot \{x, y\} \\
\mathrm{expand}(\rhd) &= \lambda x. \rhd \{x\} \\
\mathrm{expand}(\mathbf{size}) &= \lambda v.size\{v\} \\
\mathrm{expand}(\mathbf{concat}) &= \lambda x.\lambda y.concat\{x, y\} \\
\mathrm{expand}(\mathbf{iota}) &= \lambda x.iota\{x\} \\
\mathrm{expand}(\mathbf{map}) &= \lambda f.\lambda x.map\{f, x\} \\
\mathrm{expand}(\mathbf{reduce}) &= \lambda f.\lambda z.\lambda x.reduce\{z, f, x\} \\
\mathrm{expand}(\mathbf{scan}) &= \lambda f.\lambda z.\lambda x.scan\{f, z, x\}
\end{aligned} \tag{A.2}$$

$$\frac{e \rightharpoondown e'}{[e_0,\ldots,e,\ldots e_n] \rightharpoondown [e_0,\ldots,e',\ldots,e_n]}\ [\text{E-}\textsc{Arr}\text{-R}] \qquad \frac{e_1 \rightharpoondown e_1'}{e_1[e_2] \rightharpoondown e_1'[e_2]}\ [\text{E-}\textsc{Index}\text{-1-R}]$$

$$\frac{e_2 \rightharpoondown e_2'}{e_1[e_2] \rightharpoondown e_1[e_2']}\ [\text{E-}\textsc{Index}\text{-2-R}]$$

$$\frac{e \rightharpoondown e'}{(e_0\ldots,e,\ldots,e_n) \rightharpoondown (e_0,\ldots,e',\ldots,e_n)}\ [\text{E-}\textsc{Tup}\text{-R}]$$

$$\frac{e_1 \rightharpoondown e_1'}{e_1\ e_2 \rightharpoondown e_1'\ e_2}\ [\text{E-}\textsc{App}\text{-1-R}] \qquad \frac{e_2 \rightharpoondown e_2'}{v_1\ e_2 \rightharpoondown v_1\ e_2'}\ [\text{E-}\textsc{App}\text{-2-R}]$$

$$\frac{e_1 \rightharpoondown e_1'}{\textbf{let}\ p = e_1\ \textbf{in}\ e_2 \rightharpoondown \textbf{let}\ p = e_1'\ \textbf{in}\ e_2}\ [\text{E-}\textsc{Let}\text{-1-R}]$$

$$\frac{e_1 \rightharpoondown e_1'}{\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \rightharpoondown \textbf{if}\ e_1'\ \textbf{then}\ e_2\ \textbf{else}\ e_3}\ [\text{E-}\textsc{If}\text{-1-R}]$$

Figure A.7: The rules that allow rewrites of sub-expressions.

Appendix B

Summary of the Extended π-Calculus

$P, Q, S, \ldots ::=$			**Process**
	$\mid \ \mathbf{0}$		Null process
	$\mid \ P \mid Q$		Parallel composition
	$\mid \ !P$		Replicated process
	$\mid \ \nu a.P$		Restriction on name
	$\mid \ b(\vec{x}\,).P$		Receive action
	$\mid \ \bar{b}\langle N\rangle.P$		Send action
	$\mid \ \bar{b}{:}\,\langle N\rangle\,.P$		Broadcast action
	$\mid \ I(\vec{x}\,)$		Process identifier
	$\mid \ [M \bowtie N]\ P, Q$		Conditional
$L, M, N, \ldots ::=$			**Term**
	$\mid n$		Number
	$\mid b$		Name or composite name
	$\mid x$		Variable
	$\mid M \odot N$		Arithmetic operation
$b ::=$			
	$\mid a$		Name
	$\mid a \cdot M_1 \cdot \ \ldots \ \cdot M_n$	$(n > 0)$	Composite name
$A, B, C, \ldots ::=$			**Extended process**
	$\mid P$		Process
	$\mid A \mid B$		Parallel composition
	$\mid \nu a.A$		Restriction on name
	$\mid \nu x.A$		Restriction on variable
	$\mid \{{}^{M}/_{x}\}$		Active substitution

Figure B.1: The extended syntax of processes and terms, with broadcast sending and composite names

$$\textsc{Comm} \qquad \overline{b}\langle x\rangle.P \mid b(x).Q \xrightarrow{\vartriangle} P \mid Q$$

$$\textsc{Broad} \qquad \overline{b}\colon\langle x\rangle.Q \mid b(x).P_1 \mid \cdots \mid b(x).P_n \xrightarrow{:b} Q \mid P_1 \mid \cdots \mid P_n$$

$$\textsc{Par} \quad \frac{A \xrightarrow{\vartriangle} A'}{A \mid B \xrightarrow{\vartriangle} A' \mid B} \qquad \text{B-}\textsc{Par} \quad \frac{A \xrightarrow{:b} A' \qquad B \updownarrow_b}{A \mid B \xrightarrow{:b} A' \mid B}$$

$$\textsc{Res} \quad \frac{A \xrightarrow{q} A' \qquad q \neq :u}{\nu u.A \xrightarrow{q} \nu u.A'} \qquad \text{B-}\textsc{Res} \quad \frac{A \xrightarrow{:b} A'}{\nu b.A \xrightarrow{\vartriangle} \nu b.A'}$$

$$\textsc{Struct} \quad \frac{A \xrightarrow{q} A'}{B \xrightarrow{q} B'} \quad \text{if } A \equiv B \text{ and } A' \equiv B'$$

$$\textsc{Then} \qquad [M \bowtie N]\ P, Q \xrightarrow{\vartriangle} P \quad \text{if } M \bowtie N$$

$$\textsc{Else} \qquad [M \bowtie N]\ P, Q \xrightarrow{\vartriangle} Q \quad \text{if } M \not\bowtie N$$

Figure B.2: The extended reduction rules of extended processes, with new rules for broadcasting and normal communication over composite names. Here, q is either $\vartriangle$ or some $:b$.

Rename	$A \equiv A'$ by α-conversion
Par-$\mathbf{0}$	$A \mid \mathbf{0} \equiv A$
Par-A	$A \mid (B \mid C) \equiv (A \mid B) \mid C$
Par-C	$A \mid B \equiv B \mid A$
Repl	$!P \equiv P \mid !P$
New-$\mathbf{0}$	$\nu n.\mathbf{0} \equiv \mathbf{0}$
New-C	$\nu u.\nu v.A \equiv \nu v.\nu u.A$
New-Par	$A \mid \nu u.B \equiv \nu u.(A \mid B)$ when $u \notin fv(A) \cup fn(A)$
Alias	$\nu x.\{^M/_x\} \equiv \mathbf{0}$
Subst	$\{^M/_x\} \mid A \equiv \{^M/_x\} \mid A\{^M/_x\}$
Rewrite	$\{^M/_x\} \equiv \{^N/_x\}$ when $M = N$

Figure B.3: The structural congruence rules for Eπ. Notice the usage of equality on terms, which applies to both names and numbers. [14, 23]