Program Equivalences for Concurrency Abstractions in a Concurrent Lambda Calculus with Buffers, Cells and Futures.

Martina Willig

March 23, 2009

in partial fulfilment of the requirements for the degree of
Bachelor of Science in Computer Science
at the Goethe Universität Frankfurt
Department of Computer Science and Mathematics

Thesis Supervisor:
Prof. Dr. Manfred Schmidt-Schauß
Professur für Künstliche Intelligenz und Softwaretechnologie

Danksagung

Acknowledgments

Ich bedanke mich bei allen Menschen, die mich beim Schreiben dieser Arbeit unterstützt haben. Mein Dank geht insbesondere an Dr. David Sabel für seine hervorragend engagierte Betreuung, die vielen anregenden Diskussionen und guten Ratschläge, ohne die diese Arbeit nicht möglich gewesen wäre.

Besonderer Dank gilt Herrn Prof. Dr. Schmidt-Schauß für seine außerordentlich wertvollen Vorlesungen, die mein Interesse an der Theorie der Programmiersprachen geweckt haben, und mich immer wieder aufs Neue motivierten, und der mich schließlich in seine Arbeitsgruppe aufnahm.

Ich danke auch meiner Schwester Tanja, meinem Partner Ingo und meinen Freunden Markus und Daniel, für ihre Unterstützung während meines Studiums.

Erklärung
Declaration

Ich versichere hiermit, dass ich vorliegende Bachelorarbeit selbstständig und ohne Zuhilfenahme anderer Quellen oder Hilfsmittel, als der in der Arbeit angegebenen, verfasst, sowie noch nicht anderweitig für Prüfungszwecke vorgelegt habe.

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Frankfurt am Main, March 23, 2009

Martina Willig

Abstract

Various concurrency primitives had been added to functional programming languages in different ways. In Haskell such a primitive is a MVar, joins are described in JoCaml and AliceML uses futures to provide a concurrent behaviour. Despite these concurrency libraries seem to behave well, their equivalence between each other has not been proven yet. An expressive formal system is needed. In [SSNSS08] a universal calculus for concurrency primitives was defined known as the typed *lambda calculus with futures*. There, equivalence of processes had been proved. An encoding of simple one-place buffers had been worked out. This bachelor's thesis is about encoding more complex concurrency abstractions in the lambda calculus with futures and proving correctness of its operational semantics. Given the new abstractions, we will discuss program equivalence between them. Finally, we present a library written in Haskell that exposes futures and our concurrency abstractions as a proof of concept.

Contents

1. Introduction

In the last decade functional programming languages became extremely popular. Today the demand for features of functional programming languages is growing enormously. On the one hand, this is reflected by the ongoing integration of functional features into object-oriented languages like *Scala*[1] in the case of Java or $C\omega$[2] and F#[3] respectivly for C# .

On the other hand this shift towards functional languages can be observed in the growing number of users of languages like Haskell or OCaml. These provide by design the benefits of pure functional languages [Bac78]. Some of these benefits are

- *No Side Effects*: An expression in a pure functional language is a mathematical function and function application. Its value depends only on its arguments. For example, the function $f(x)$ will produce the same result for a given argument x at any time. This avoids saving the program state and mutable data. Because there is no need for changing the value of already calculated computations, functional programming languages do not have any side effects.

- *Type Safety*: The type system prevents type errors at run time because typing mistakes are already found by the type-checker during the compilation process.

- *Program clarity [Bac78]*: Programs written in a functional language are intuitive, lean and easy to read.

[1]The object-oriented Java is mixed with the functional features of ML into *Scala*, developed at École polytechnique fédérale de Lausanne

[2]$C\omega$ (formerly known as Polyphonic C#, developed at INRIA) is based on the join calculus and developed at Microsoft Research [Res09a].

[3]F# stems from the ML family of languages and has a core language compatible with that of OCaml, Microsoft Research [Res09b].

- *Provableness*: Functional languages are based on a mathematical axiomatic system - a calculus - which is a powerful tool for providing program correctness.
- *Abstraction*: The semantics of functional languages is as abstract as possible.
- *Platform Independence*: This follows from abstraction: the higher the abstraction the better the integration of a language on different architectures.

Functional programming languages are very powerful but programmers cannot write real world applications with this pure but limited approach. Real world applications usually need data access, input/output, multiple processes which interchange their values and have connections to other systems. Furthermore they may be distributed themselves. Therefore common functional programming languages had been extended by libraries empowering them in one particular task like for example concurrency, parallelism or communication. But at the same time this also weakens them: In the case of concurrency indeterminism is not always avoidable. Think of two processes that try to write to a storage location in memory in parallel. One of them had to be chosen as the first writer so the other process fails to write and must try again later. This example breaks the rule of absence of side effects and conflicts with the original idea of the functional approach and its powerful features. But although these unfavourable properties are not avoidable, they can be reduced to a small set of cases in which they occur. This requires the extensions to be written in a formal way. Once we have a formalism we can use it to verify that the semantics satisfy a particular specification and observe how programs behave.

Concurrent behaviour is especially interesting, since its programming model is essential in modern programming. There are many different ways to add concurrent behaviour to a programming language. The concurrency libraries differ from implementation to implementation. For example in Haskell's `Control.Concurrent` module a synchronisation between two processes is provided by simple *semaphores*[4] while in JoCaml[5] the same be-

[4]Semaphores are called `MVars` here. They are simple lock variables. See [PGF96, Pey09].

[5]JoCaml is a derivate of the join calculus. See [INR09] for further reading.

haviour is achieved by *joins*[6].

A basic formalism that describes the syntax and semantics of a language supporting concurrent behaviour had been developed in [NSS06]. It is called the *lambda calculus with futures* and its speciality is a paradigm descended from the field of parallel computing called a *future*. This calculus has been extended numerous times with new syntactical and semantical elements. Each extension means creating a new calculus. Thus, the extensions form a chain of calculi differing only in a few minor points. Within the development of this chain an observational semantics for the lambda calculus with futures had been introduced and improved. The observational semantic turned out to be an excellent proving tool for concurrent calculi.

We refer to the calculus called $\lambda^\tau(\mathsf{fchb})$, which was lately recalled in [SSNSS09]. So far, $\lambda^\tau(\mathsf{fchb})$ is a sleek calculus that models a minimal need of concurrency paradigms, whereas it is an extremely expressive method of proof for correctness of program transformations and calculi translations.

In this thesis we extend the lambda calculus with futures by more complex concurrency primitives (Chapter 2.2). We translate them into terms of $\lambda^\tau(\mathsf{fchb})$. In chapter 3 we show properties of these concurrency abstractions and correct transformation rules making use of the powerful observational semantics. We also show program equivalences between new concurrency abstractions. In doing so, we stay at the level of $\lambda^\tau(\mathsf{fchb})$. These program equivalences can be used in the proofs of translations between two calculi in the future. Finally, we implement the primitives encodings in Haskell (Chapter 4) using the `Control.Concurrent` library based on [PGF96]. By doing so we make them comparable to Haskell's implementation.

[6]Joins are a primitive for synchronising processes.

2. A Formalism For Concurrency

A formal system for concurrent behaviour must fulfil the requirements of concurrent programming issues. We give an overview of the main aspects in this section.

2.1 Objectives In Concurrent Programs

```
S: let x = 0
   in x:=x+1; x:=x+2;
   print x
```

Figure 2.1: A sequential program.

```
P: let x = 0
   in x:=x+1 | x:=x+2;
   print x
```

Figure 2.2: A parallel program.

Safety. While in sequential programming the order of execution is deterministic, in concurrent programs it is not. This arises some difficulties for instance if two or more independent processes access a shared memory. In this case the order of execution can affect the result of the program. To illustrate this, consider program S (Fig. 2.1) and P (Fig. 2.2) that manipulate data. The printing command in the sequential programm S will output '3', every time the code is executed. In contrast P can produce '1', '2' or '3' as the output for `print`. This behaviour is called *interference*. Each part in the code where an assignment is made for x represents a *critical region* because it uses the shared memory variable x whose state can change. Critical regions are a source for interference and can be avoided by synchronisation mechanisms. If a concurrent program never enters a bad state, then it fulfills the *safety* property.

Liveness. *Deadlocks* arise from the situation of two threads blocking each other. In Fig. (2.3) both processes await the access permission of the variable held by the other process. They will wait forever. *Livelocks* arise from a cycle

```
P: acquire A; acquire B; compute; release B; release A
Q: acquire B; acquire A; compute; release A; release B
```

Figure 2.3: A Deadlock situation, in [Rep99]

in the dependence of ressources of two processes, not blocking each other. In

```
        P                          Q
1: acquire A;             1: acquire B;
   if (B is held):           if (A is held):
      release A; goto 1         release B; goto 1
   else                      else
      acquire B;                acquire A;
   compute;                  compute;
   release B;                release A;
   release A                 release B
```

Figure 2.4: A Livelock situation, in [Rep99]

Fig. (2.4) the processes P and Q are both retrying to evaluate the state of A (B) in the if condition. Here, the code is cyclic because the condition will never evaluate to `True`. The liveness property asserts that a program will always reach a desirable state: convergence (termination) or divergence.

Fairness. Assume a situation where a scheduler is only based on priority. There may always exist processes with a higher priority than a given process p. Thus p will always be delayed. This is called *starvation.* In concurrent programs processes and threads should be executed in a fair order. Every process must have its chance to execute. This presumes the fairness property.

These objectives define a part of the measure of power that the semantic of $\lambda^{\tau}(\mathsf{fchb})$ needs.

2.2 The $\lambda^{\tau}(\textsf{fchb})$ Calculus

2.2.1 Main Concepts

Processes and Components. A program in $\lambda^{\tau}(\textsf{fchb})$ consists of processes. A process is a collection of components combined into parallel computation. Components are either threads or handles, cells or buffers. A component is a sequentially computated expression. Threads are *lazy* or *eager*: A lazy thread evaluates its value by need, an eager thread immediately evaluates. Lazy and eager threads can be *mixed* together. We do not distinguish between structural congruent processes: The order of introducing threads in a process does not affect any order of evaluation in a parallel computation. Processes synchronise on future values: A successful thread evaluates to a proper value and binds it to a future. Processes can observe other processes on unbound names. Bound names are hidden from observation via the keyword (ν).

Synchronisation: Futures and Handles. Processes synchronise their values via futures. Future values are lazily evaluated so they explicitly suspend the computation. Each future object is associated with a background thread that computes the future value. As long as this expression has not been evaluated, the value of the future is unknown. Whenever an unknown future is accessed the computation will suspend on this future. Once the value has been evaluated the computation resumes. A *handle* is a component that points to an unevaluated future and computes its value on demand. Therefore, handles are used to associate a value to a future. They provide a synchronisation mechanism for processes.

The $\lambda^{\tau}(\textsf{fchb})$'s evaluation strategy is a strict one, except it is explicitly expressed by the use of futures and lazy threads. $\lambda^{\tau}(\textsf{fchb})$ is *typed.* The language also models case-expressions and data constructors that include also tuples such as pairs and lists. Finally, **reference cells** point to a location in memory, to mutable data.

$$
\begin{array}{rcl}
\tau \in \mathit{Type} & ::= & \mathsf{unit} \mid \mathsf{ref}\ \tau \mid \tau \to \tau \mid \kappa(\tau_1, \ldots, \tau_{ar(\kappa)}) \mid \mathsf{buf}\ \tau \\
c \in \mathit{Const} & ::= & \mathbf{unit} \mid \mathbf{cell}^\tau \mid \mathbf{thread}^\tau \mid \mathbf{lazy}^\tau \mid \mathbf{handle}^\tau \mid \mathbf{newBuf}^\tau \mid \mathbf{get}^\tau \\
\pi \in \mathit{Pat} & ::= & k^\tau(x_1, \ldots, x_{ar(k)}) \\
e \in \mathit{Exp} & ::= & x \mid c \mid \lambda x.e \mid e_1\ e_2 \mid \mathbf{exch}(e_1, e_2) \mid k^\tau(e_1, \ldots, e_{ar(k)}) \mid \\
 & & \mathbf{case}_\kappa\ e\ \mathbf{of}\ \pi_1 \Rightarrow e_1 \mid \ldots \mid \pi_m \Rightarrow e_m \mid \mathbf{put}(e_1, e_2) \\
v \in \mathit{Val} & ::= & x \mid c \mid \lambda x.e \mid k^\tau(v_1, \ldots, v_{ar(k)}) \\
p \in \mathit{Proc} & ::= & p_1 \mid p_2 \mid (\nu x)p \mid x\,\mathsf{c}\,v \mid x \Leftarrow e \mid x \overset{susp}{\Longleftarrow} e \mid y\,\mathsf{h}\,x \mid y\,\mathsf{h}\ \bullet \mid \\
 & & x\,\mathsf{b}\ - \mid x\,\mathsf{b}\,v
\end{array}
$$

Figure 2.5: Syntax description of $\lambda^\tau(\mathsf{fchb})$

$$
\begin{array}{lll}
p_1 \mid p_2 \ \equiv\ p_2 \mid p_1 & (p_1 \mid p_2) \mid p_3 \ \equiv\ p_1 \mid (p_2 \mid p_3) & \\
(\nu x)(\nu y)p \equiv (\nu y)(\nu x)p & (\nu x)(p_1) \mid p_2 \equiv (\nu x)(p_1 \mid p_2) & \text{if } x \text{ is not free in } p_2
\end{array}
$$

Figure 2.6: Structural congruence of processes

2.2.2 Syntax

The syntax consists of two layers: Expressions $e \in \mathit{Exp}$ which sequentially compute λ-expressions and case-expressions, and processes $p \in \mathit{Proc}$ which compose components in parallel. Constants or higher-order functions are: $\mathbf{thread}^\tau$ introduces a new eager thread. $\mathbf{lazy}^\tau$ introduces a new lazy thread. $\mathbf{cell}^\tau$ introduces a reference cell and $\mathbf{handle}^\tau$ introduces a new handle. $\mathbf{exch}(e_1, e_2)$ is the operation for exchanging cell values. Values are variables, λ-expressions or data constructors. A cell is the smallest unit for data storage. A buffer is like a cell with a synchronisation ability. $\mathbf{newBuf}^\tau$ creates an empty buffer that swells in a new process; $\mathbf{get}^\tau$ takes the contents of a buffer or suspends on an empty buffer until it holds a value and $\mathbf{put}(b, v)$ is a binary operator that puts a value v into an empty buffer b or suspends until the buffer becomes empty. Processes p are composed in parallel from smaller components $p_1 \mid p_2$, whereas the operator $\mid$ does not define any computation order. (Fig. 2.6). The table (2.1) lists all distinguishable forms of components. All processes satisfy the distinct variable convention. That means, new binders use fresh variables. Thus, if values with bound variables had been copied by β-CBV(ev), CASE.BETA(ev) or FUT.DEREF(ev), they must be α-renamed before applying the next rule. We assume that the

$x{\Leftarrow}e$	is an eager thread. An eager thread binds a future x to the value of expression e unless it diverges or suspends; x is called a *concurrent future.*
$x \overset{susp}{\Longleftarrow} e$	is a lazy thread. A lazy thread is a suspended computation that starts once the value of x is needed elsewhere; x is called a *lazy future.*
$y\,\mathsf{h}\,x$	is a handle. A handle associates y to a future x, so that y can be used to assign a value to x; x is called a *handled future.*
$y\,\mathsf{h}\,\bullet$	is a used handle. A used handle has already bound its associated future to a value.
$x\,\mathsf{c}\,v$	associates the value v to the cell x.
$x\,\mathsf{b}\,-$	is a new empty buffer.
$x\,\mathsf{b}\,v$	is a buffer containing the value v.

Table 2.1: Forms of components

distinct variable convention holds: All free variables are distinct from bound variables and bound variables are pairwise distinct.

Definition 2.2.1 (Variable introduction)**.** A process $p \equiv D[p']$ introduces a variable x for p' such that p' is $x{\Leftarrow}v, x\,\mathsf{c}\,v, x \overset{susp}{\Longleftarrow} e, x\,\mathsf{h}\,v, y\,\mathsf{h}\,x, x\,\mathsf{h}\,\bullet$ or $x\,\mathsf{b}\,v$.

Introduced variables are called process variables.

Definition 2.2.2 (Well-formedness)**.** A process is well-formed, if it does not introduce any variable twice.

An empty buffer x is denoted as $x\,\mathsf{b}\,-$. A buffer x that contains a value v is represented by $x\,\mathsf{b}\,v$. Finally, the syntax of $\lambda^\tau(\mathsf{fchb})$ includes *patterns* $\pi \in Pat$ to distinguish between different data constructors in a case-expression.

Typing. The finite set of data and type constructors $\Sigma = (K, D)$ consists of type constructors $\kappa \in K$ and data constructors $k \in D$. For all variables x there is a unique type $\Gamma(x)$. τ denotes a monomorphic type. unit is a monomorphic type representing the empty value and $\mathsf{ref}\ \tau$ is a monomorphic type that denotes a reference to a memory location. Monomorphic types are polymorphic types without type variables. Each data constructor has a unique polymorphic type $upt(k)$. $upt(k)$ is of the form $\widehat{\tau}_1 \to \cdots \to \widehat{\tau}_{ar(k)} \to \kappa(\alpha_1, \ldots, \alpha_{ar(k)})$. $ar()$ is the arity of k. Polymorphic types are denoted as

$$x : \Gamma(x) \quad \textbf{unit} : \mathsf{unit} \quad \textbf{cell}^{\tau} : \alpha{\to}\mathsf{ref}\ \alpha$$
$$\textbf{thread}^{\tau} : (\alpha{\to}\alpha){\to}\alpha \qquad \textbf{lazy}^{\tau} : (\alpha{\to}\alpha){\to}\alpha \qquad \textbf{get}^{\tau} : \mathsf{buf}\ \alpha \to \alpha$$
$$\textbf{handle}^{\tau} : (\alpha{\to}(\alpha{\to}\mathsf{unit}){\to}\beta){\to}\beta \quad \textbf{newBuf}^{\tau} : \mathsf{unit} \to \mathsf{buf}\ \alpha$$

$$\frac{x : \tau_1 \quad e : \tau_2}{(\lambda x.e) : \tau_1 \to \tau_2} \qquad \frac{e_1 : \tau_1 \to \tau_2 \quad e_2 : \tau_1}{(e_1\ e_2) : \tau_2} \frac{e_1 : \mathsf{ref}\ \tau \quad e_2 : \tau}{\textbf{exch}(e_1, e_2) : \tau} \qquad \frac{e_1 : \mathsf{buf}\tau \quad e_2 : \tau}{\textbf{put}(e_1, e_2) : \mathsf{unit}}$$

$$\frac{k \in \mathcal{D}_\kappa \quad e_1 : \tau_1 \quad \ldots \quad e_{ar}(k) : \tau_{ar(k)} \quad k : \tau_1 \to \ldots \to \tau_{ar(k)} \to \kappa(\tau'_1, \ldots, \tau'_k)}{k(e_1, \ldots, e_{ar(k)}) : \kappa(\tau'_1, \ldots, \tau'_k)}$$

$$\frac{\mathcal{D}(\kappa) = \{k_1, \ldots, k_n\} \quad e : \kappa(\tau'_1, \ldots, \tau'_k) \quad \forall i = 1 \ldots n :\ e_i : \tau \quad \forall i, j : x_{i,j} : \tau_{i,j}}{\dfrac{\forall i = 1 \ldots n : k_i : \tau_{i,1} \to \ldots \to \tau_{i,m_i} \to \kappa(\tau'_1, \ldots, \tau'_k)}{(\textbf{case}_\kappa\ e\ \textbf{of}\ (k_i(x_1, \ldots, x_{ar(k_i)})\ \Rightarrow e_i)^{i=1\ldots n}) : \tau}}$$

Figure 2.7: Typing rules for expressions

$$\frac{p_1 : wt \quad p_2 : wt}{p_1 \mid p_2 : wt} \qquad \frac{x{:}\tau \quad e{:}\tau}{x{\Leftarrow}e{:}wt} \qquad \frac{x{:}\tau \quad e{:}\tau}{x \overset{susp}{\Longleftarrow} e{:}wt} \qquad \frac{x{:}\mathsf{ref}\ \tau \quad v{:}\tau}{x\ \mathsf{c}\ v{:}wt} \qquad \frac{p{:}wt}{(\nu x)p{:}wt}$$

$$\frac{}{y\ \mathsf{h}\ \bullet : wt} \qquad \frac{x :\ \tau \quad y :\ \tau\ \to\ \mathsf{unit}}{y\ \mathsf{h}\ x : wt} \qquad \frac{}{x\ \mathsf{b} - : wt} \qquad \frac{x :\ \mathsf{buf}\ \tau \quad v :\ \tau}{x\ \mathsf{b}\ v : wt}$$

Figure 2.8: Typing of processes

$\widehat{\tau} \in \mathit{PolyType} ::= \alpha \mid \mathsf{unit} \mid \mathsf{ref}\ \widehat{\tau} \mid \widehat{\tau} \to \widehat{\tau} \mid \kappa(\alpha_1, \ldots, \alpha_{ar(k)})$ where α belongs to a finite set of type variables. Monomorphic instances τ of polymorphic types $\widehat{\tau}$ are denoted as $\tau \preceq \widehat{\tau}$. The type system allows n-tuples $\langle v_1, \ldots, v_n \rangle$ of all possible types $\tau_1 \times \cdots \times \tau_n$.

Example 2.2.1. Let $\mathsf{Bool} \in \mathrm{K}$ be a type constructor. Then $D_{Bool} = \{\mathsf{True}, \mathsf{False}\}$ is the set of data constructors for Bool. Bool is of arity 0. True and False are monomorphic types of arity 0.

Example 2.2.2. We define a polymorphic type for lists. Let $\mathsf{List} \in \mathrm{K}$ be the type constructor of polymorphic lists. Then we define two data constructors, one for the empty list $\mathsf{nil} \in \mathrm{D}(\mathsf{List})$ and one for a list holding at least one

α-typed element $\mathsf{cons} \in \mathrm{D}(\mathsf{List})$ with $\mathrm{upt}(\mathsf{nil}) = \mathsf{List}(\alpha)$ and $\mathrm{upt}(\mathsf{cons}) = \alpha \rightarrow \mathsf{List}(\alpha) \rightarrow \mathsf{List}(\alpha)$. List is of arity 1, nil of arity 0 and cons of arity 2.

Process components have to be well-typed, denoted by $p : wt$. Type assignments for expressions are written $e : \tau$. Typing Rules for expressions and processes are shown in the Fig. 2.7 and Fig. 2.8.

Remark. We omit type lables for constants and constructors if they do not play an important role. Mostly in the proofs in chapter 3, typing remains out of consideration without affecting the result.

With all these syntactical elements we can construct programs in $\lambda^\tau(\mathsf{fchb})$. For illustration consider the Example 2.2.3.

Example 2.2.3. A syntactically valid program in $\lambda^\tau(\mathsf{fchb})$ is for example

$$\underbrace{\underbrace{(\nu z)(\nu h)(\nu b)(\underbrace{z \Leftarrow \textbf{get}^\tau\, b}_{\text{a thread}} \mid \underbrace{h\, \mathsf{h}\, f}_{\text{a handle}} \mid \underbrace{b\, \mathsf{b}\, 4}_{\text{a buffer}})}_{\text{process 1}} \mid \underbrace{(\nu y)(\underbrace{y \overset{susp}{\Longleftarrow} e}_{\text{a susp. thread}})}_{\text{process 2}}}_{\text{a program}}$$

where e is an expression in *Exp*.

2.2.3 Abbreviations

While describing the semantics and especially when encoding new primitives, we use the following syntactical abbreviations:

- $\mathsf{let}\ x_1{=}e_1, \ldots, x_n{=}e_n\ \mathsf{in}\ e$ is shorthand for $\mathsf{let}\ x_1{=}e_1\ \mathsf{in}\ , \mathsf{let}\ x_2{=}e_2\ \mathsf{in}\ , \ldots, \mathsf{let}\ x_n{=}e_n\ \mathsf{in}\ e$ and this is shorthand for $(\lambda x_n.(\lambda x_{n-1}.(\ldots(\lambda x_1.e)e_1\ldots))e_{n-1})e_n$
- $e_1; e_2$ is the sequencing computation and is equal to $(\lambda\_.(e_2))e_1$
- Patterns (guards) π in abstractions $\lambda\pi.e$ are used instead of $\lambda x.\textbf{case}\ x\ \textbf{of}\ \pi \Rightarrow e$.
- $\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ is shorthand for $\textbf{case}\ e\ \textbf{of}\ \mathsf{True} \Rightarrow e_1 \mid \mathsf{False} \Rightarrow e_2$
- $\mathsf{wait}\ e$ is used instead of $\mathsf{if}\ e\ \mathsf{then}\ \mathsf{True}\ \mathsf{else}\ \mathsf{True}$
- $\_$ stands for an arbitrary fresh variable.
- $\bot$ is a must-divergent computation
- $(\nu x_1, \ldots, x_n)$ means that $x_1, x_2 \ldots x_n$ are bound variables.

Concerning the buffer primitive we write $\textbf{newBuf}\ v$ for creating a new buffer containing the value v. So $\textbf{newBuf}\ v$ equates to $\mathsf{let}\ b = \textbf{newBuf}\ \mathsf{unit}\ \mathsf{in}\ \textbf{put}(b, v); b$. In our proofs in chapter 3 we use $\xrightarrow{*}$ for the reflexive-transitive closure of $\rightarrow$ and $\xrightarrow{+}$ for the transitive closure. We often label the arrow by the applied reduction, e.g. $p \xrightarrow{\beta\text{-CBV}(ev)} p'$.

2.2.4 Operational Semantics

$$
\begin{array}{ll}
\text{ECs} & E ::= x \Leftarrow \widetilde{E} \\
 & \widetilde{E} ::= [\,] \mid \widetilde{E}\, e \mid v\, \widetilde{E} \mid \textbf{exch}(\widetilde{E}, e) \mid \textbf{exch}(v, \widetilde{E}) \\
 & \quad \mid \textbf{case}\ \widetilde{E}\ \textbf{of}\ (\pi_i \Rightarrow e_i)^{i=1\ldots n} \mid k(v_1, \ldots v_{i-1}, \widetilde{E}, e_{i+1}, \ldots, e_n) \\
 & \quad\ \textbf{put}(\widetilde{E}, e) \mid \textbf{put}(v, \widetilde{E}) \\
\text{Future ECs} & F ::= x \Leftarrow \widetilde{F} \\
 & \widetilde{F} ::= \widetilde{E}[[\,]\, v] \mid \widetilde{E}[\textbf{exch}([\,], v)] \mid \widetilde{E}[\textbf{case}\ [\,]\ \textbf{of}\ (\pi_i \Rightarrow e_i)^{i=1\ldots n}] \\
 & \quad\ \widetilde{E}[\textbf{put}([\,], v)] \mid \widetilde{E}[\textbf{get}[\,]] \\
\text{Process ECs} & D ::= [\,] \mid p \mid D \mid D \mid p \mid (\nu x) D
\end{array}
$$

Figure 2.9: Evaluation contexts

Evaluation Contexts and Strategies. Reduction rules apply in an evaluation context (Fig. 2.9). A context is a process or expression with a single occurrence of the hole marker $[\,]$.

Definition 2.2.3 (Substitution)**.** Let γ be a context, and η be a term that can be plugged into its hole, then the result of replacing the hole $[\,]$ in γ by η is written as $\gamma[\eta]$.

A restriction is that the hole marker cannot occur at positions that are reserved for variable introduction or in a cell-value v, except it is an abstraction $v = \lambda x.e$. In this case the hole may occur in e. A context is called *flat*, if the hole does not occur below a λ-binder or a case-alternative, otherwise the context is *deep*.

Which contexts are affected by a reduction rule is defined by the applied evaluation strategy. The strategy defines the validity of a reduction rule in a specific context. The standard evaluation strategy of $\lambda^{\tau}(\mathsf{fchb})$ is denoted by ev. Strategy ev permits reduction in ECs except for the rules (FUT.DEREF(ev)) and (LAZY.TRIGGER(ev)). They apply in future contexts F, using ev as the strategy.

Contexts EC encode the standard call-by-value, left-to-right reduction strategy. Call-by-value means that all arguments of a function are evaluated before function application is evaluated. Futures ECs are used to trigger suspended threads or while dereferencing a futures value. Future ECs encode a data-driven synchronisation: If $x \Leftarrow e$ is a future then another thread will

(β-$\textsc{cbv}(\mathsf{ev})$)	$E[(\lambda y.e)\, v] \rightarrow E[e[v/y]]$
($\textsc{thread.new}(\mathsf{ev})$)	$E[\mathbf{thread}^{\tau}\, v] \rightarrow (\nu z)(E[z] \mid z \Leftarrow v\, z)$
($\textsc{fut.deref}(\mathsf{ev})$)	$F[x] \mid x \Leftarrow v \rightarrow F[v] \mid x \Leftarrow v$
($\textsc{handle.new}(\mathsf{ev})$)	$E[\mathbf{handle}^{\tau}\, v] \rightarrow (\nu z)(\nu z')(E[v\, z\, z'] \mid z' \,\mathsf{h}\, z)$
($\textsc{handle.bind}(\mathsf{ev})$)	$E[x\, v] \mid x \,\mathsf{h}\, y \rightarrow E[\mathbf{unit}] \mid y \Leftarrow v \mid x \,\mathsf{h}\, \bullet$
($\textsc{cell.new}(\mathsf{ev})$)	$E[\mathbf{cell}^{\tau}\, v] \rightarrow (\nu z)(E[z] \mid z \,\mathsf{c}\, v)$
($\textsc{cell.exch}(\mathsf{ev})$)	$E[\mathbf{exch}(z, v_1)] \mid z \,\mathsf{c}\, v_2 \rightarrow E[v_2] \mid z \,\mathsf{c}\, v_1$
($\textsc{lazy.new}(\mathsf{ev})$)	$E[\mathbf{lazy}^{\tau}\, v] \rightarrow (\nu z)(E[z] \mid z \overset{susp}{\Longleftarrow} v\, z)$
($\textsc{lazy.trigger}(\mathsf{ev})$)	$F[x] \mid x \overset{susp}{\Longleftarrow} e \rightarrow F[x] \mid x \Leftarrow e$
($\textsc{case.beta}(\mathsf{ev})$)	$E[\mathbf{case}\ k_j(v_1, \dots, v_{ar(k_j)})\ \mathbf{of}\ (k_i(x_1, \dots, x_{ar(k_i)}) \rightarrow e_i)^{i=1\dots n}] \rightarrow E[e_j[v_1/x_1, \dots, v_{ar(k_j)}/x_{ar(k_j)}]]$
($\textsc{buff.new}(\mathsf{ev})$)	$E[\mathbf{newBuf}^{\tau}\ v] \rightarrow (\nu x)(E[x] \mid x \,\mathsf{b}\, -)$
($\textsc{buff.put}(\mathsf{ev})$)	$E[\mathbf{put}^{\tau}(x, v)] \mid x \,\mathsf{b}\, - \rightarrow\ E[\mathbf{unit}] \mid x \,\mathsf{b}\, v$
($\textsc{buff.get}(\mathsf{ev})$)	$E[\mathbf{get}^{\tau}\ x] \mid x \,\mathsf{b}\, v \rightarrow\ E[v] \mid x \,\mathsf{b}\, -$

Figure 2.10: Reduction rules . One-step reduction relation of $\lambda^{\tau}(\mathsf{fchb})$ is denoted by $\rightarrow$ or $\xrightarrow{\mathsf{ev}}$

use x like it was a value until it needs the computed value of x. At this point of time the thread must evaluate the future. Finally, process ECs D are contexts for evaluating process components, since programs in our language are composed by processes.

Example 2.2.4. A reduction inside a thread $x \Leftarrow e$ means to reduce a subexpression e' in an evaluation context E where $e = E[e']$. The expression e' evaluates call-by-value in $\widetilde{E}$. In contrast, futures have their own evaluation context F because a future should only be replaced by its value if it is needed to proceed with the computation of a thread.

Reduction Rules. The reduction rules are listed in Fig. 2.10. The rule (β-$\textsc{cbv}(\mathsf{ev})$) is called call-by-value beta-reduction. In an expression $(\lambda y.e)\ v$ it replaces the bound variable y in all occurences of e by v. The rule ($\textsc{case.beta}(\mathsf{ev})$) compares a data constructor with the patterns π. If a match was found it returns the right handside of the corresponding alternative. For each higher order function there exists a reduction rule.

$(\textsc{det.put})$	$(\nu x)(E[\mathbf{put}^\tau(x,v)] \mid x\,\mathsf{b}\,-) \to (\nu x)(E[\mathbf{unit}] \mid x\,\mathsf{b}\,v)$
$(\textsc{det.get})$	$(\nu x)(E[\mathbf{get}^\tau\ x] \mid x\,\mathsf{b}\,v) \to (\nu x)(E[v] \mid x\,\mathsf{b}\,-)$
$(\textsc{det.exch})$	$(\nu x)(y{\Leftarrow}\widetilde{E}[\mathbf{exch}(x,v_1)] \mid y\,\mathsf{c}\,v_2) \to (\nu x)(y{\Leftarrow}\widetilde{E}[v_1] \mid y\,\mathsf{c}\,v_1)$
$(\textsc{cell.deref}(\mathsf{ev}))$	$p \mid y\,\mathsf{c}\,x \mid x{\Leftarrow}v \to p \mid y\,\mathsf{c}\,v \mid x{\Leftarrow}v$
$(\textsc{gc})$	$p \mid (\nu y_1)\dots(\nu y_n)p' \to p$ if p' is successful and $y_1,\dots,y_n$ contain all process variables of p'

Figure 2.11: Correct transformation rules

The rule $(\textsc{cell.new}(\mathsf{ev}))$ creates a new cell z with content v resulting in a new component $z\,\mathsf{c}\,v$. The rule $(\textsc{cell.exch}(\mathsf{ev}))$ encodes the behaviour of the exchange operation $\mathbf{exch}(z,v_1)$ that writes v_1 to the cell z and returns the previous contents of the cell in one atomic step. $(\textsc{cell.exch}(\mathsf{ev}))$ is a non-deterministic rule:

Example 2.2.5. Assume a process which creates a cell and gives access to that cell for other threads. The cell can be changed by different threads in parallel. $p{\Leftarrow}\mathsf{let}\ c = \mathbf{cell}^\tau\ x\ \mathsf{in}\ (\mathbf{thread}^\tau\ (fc); \mathbf{thread}^\tau\ (gc))$
$\xrightarrow{*} p{\Leftarrow}\mathbf{thread}^\tau\ (fz); \mathbf{thread}^\tau\ (gz) \mid z\,\mathsf{c}\,x$

The rule $(\textsc{thread.new}(\mathsf{ev}))$ spawns a new eager thread x that computes an expression e resulting in a thread component $x{\Leftarrow}e$. Note, that if the thread is recursive, x may occur in e such that $x = e$. The rule $(\textsc{lazy.new}(\mathsf{ev}))$ spawns a new suspended thread component $x \overset{susp}{\Longleftarrow} e$. The dereferencing operation of future values $(\textsc{fut.deref}(\mathsf{ev}))$ replaces a future by its value.

The rule $(\textsc{lazy.trigger}(\mathsf{ev}))$ triggers a suspended computation resulting in a non-suspended thread. The rule $(\textsc{handle.new}(\mathsf{ev}))$ creates a new handle component $y\,\mathsf{h}\,x$. The rule $(\textsc{handle.bind}(\mathsf{ev}))$ applied on $\widetilde{E}[z{\Leftarrow}yv \mid y\,\mathsf{h}\,x]$ consumes the handle y and binds x to v, resulting in a used handle and new thread $x{\Leftarrow}v$.

The rule $(\textsc{buff.new}(\mathsf{ev}))$ introduces a new buffer component $x\,\mathsf{b}\,-$.
The rule $(\textsc{buff.put}(\mathsf{ev}))$ can only be applied on an empty buffer and reduces to the empty value after writing into the buffer. The rule $(\textsc{buff.get}(\mathsf{ev}))$ can only be applied on a non-empty buffer, empties it and returns its value. $(\textsc{buff.put}(\mathsf{ev}))$ and $(\textsc{buff.get}(\mathsf{ev}))$ are non-deterministic: The identifier variable x of a buffer component is not (ν)-bound, thus it could be coincidentally altered by more than one thread. Therefore in [SSNSS08] deterministic variants are given as transformation rules (Fig. 2.2.4). Here, the

identifier is bound and simultaneous modification is excluded. (DET.PUT) and (DET.GET) are correct transformation rules [SSNSS08]. $\lambda^\tau(\mathsf{fchb})$ also has a garbage collector (GC). Once, the garbage collection transition is used, it removes all finished and unreferenced threads. (GC) is also a correct transformation.

Example 2.2.6. Using (HANDLE.BIND(ev)) the program $x \Leftarrow yv \mid y\,\mathsf{h}\,f$ reduces to $x \Leftarrow \mathsf{unit} \mid y\,\mathsf{h}\,\bullet \mid f \Leftarrow v$. Applying GC to the term removes all its components (x, f, y), since they all have finished the computation.

2.2.5 Observational Semantics

Observational semantics for $\lambda^\tau(\mathsf{fchb})$ programs allows us to reason about the correctness of transformations of stateful and concurrent computations. If we want to reason about programs in $\lambda^\tau(\mathsf{fchb})$, we need some well-defined program properties such as successfullness, convergence and correctness of programs. Once we have these terminologies we can use them to give evidence of the correctness of encodings and of new primitives from chapter 2.3 and to show equivalences of programs.

Theorem 1 (Successfullness, [SSNSS09]). *A process p is* successful *if it is well-formed and in every component $x \Leftarrow e$ of p, the identifier x is bound possibly via a chain $e \Leftarrow x_1 \mid x_1 \Leftarrow x_2 \mid \ldots \mid x_{n-1} \Leftarrow x_n$ to a non-variable value, a cell or a lazy future, a handled future, a handle or a buffer.*

Hence, in $\lambda^\tau(\mathsf{fchb})$ the computations $x \Leftarrow \lambda y.y$, $x \Leftarrow y \mid y \Leftarrow \langle x, x\rangle$ and $x \Leftarrow y \mid y\,\mathsf{c}\,z$ are successful, while $x \Leftarrow (\lambda ba.a)(y\ \mathsf{unit}) \mid y \Leftarrow (\lambda ba.a)(x\ \mathsf{unit})$ (a deadlock) and $x \Leftarrow x$ (a black hole) are ruled out.

A Notion for Equivalence. To show contextual equivalence of two processes or expressions in $\lambda^\tau(\mathsf{fchb})$, we must show first that they reduce to the same values. Secondly it must state that for all contexts, where the terms may occur in, their convergence behaviour is equal. We need a terminology that clearly defines convergency. This is done by the formal definition of may-must-convergence.

Definition 2.2.4 (May-Must Convergence,[SSNSS08]). Let p be a process, then p is *may-convergent* ($p\downarrow$) if there exists a sequence of reductions $p \rightarrow^* p'$ such that p' is successful or p is *must-convergent* ($p\Downarrow$) if all reduction successors p' of p are may-convergent. If p has no reduction descendant that succeeds then p is *must-divergent* ($p\Uparrow$). It is *may-divergent* ($p\uparrow$) if some reduction descendant of p is must-divergent. Thus,

$$p\Uparrow \Leftrightarrow \neg p\downarrow \text{ and } p\uparrow \Leftrightarrow \neg p\Downarrow.$$

We define a binary relation between processes $p_1, p_2 \in \ \mathit{Proc}$ and expressions $e_1, e_2 : \tau \ \in \mathit{Exp}$ so that:

$$\begin{aligned} p_1 \leq p_2 \Leftrightarrow \ \forall D : D[p_1]\downarrow \Rightarrow D[p_2]\downarrow \text{ and } D[p_1]\Downarrow \Rightarrow D[p_2]\Downarrow \text{ and} \\ e_1 \leq_\tau e_2 \Leftrightarrow \ \forall D : D[C[e_1]]\downarrow \Rightarrow D[C[e_2]]\downarrow \text{ and } e_1, e_2 : D[C[e_1]]\Downarrow \Rightarrow D[C[e_2]]\Downarrow \end{aligned}$$

The relation $\leq$ is called *contextual preorder*. $\leq_\tau$ is the contextual preorder with consideration of types. These definitions lead us to the contextual equivalence:

Definition 2.2.5 (Contextual Equivalence,[SSNSS08])**.** Let $p_1, p_2 \in \mathit{Proc}$ then $p_1 \leq p_2$ and $p_2 \leq p_1 \ \Rightarrow \ p_1 \sim p_2$. We say that p_1 and p_2 are *contextual equivalent.*

Transformations. In the correctness proofs of the encoding of new primitives we will use correct program transformations. Thus, we need a formal definition of *correct* transformations.

Definition 2.2.6 (Correctness of Transformations, [SSNSS08])**.** A transformation t is correct if and only if $p, p' \in t \Rightarrow p \sim p'$.

All reduction rules in (Fig. 2.10) of $\lambda^\tau(\mathsf{fchb})$ except ($\textsc{cell.exch}(\mathsf{ev})$) as well as the transformation rules in (Fig. 2.2.4) are correct. The proofs can be found in [SSNSS08].

Translations. Our proofs of correct transformations in $\lambda^\tau(\mathsf{fchb})$ are a base for a proof of a translation $T : \lambda' \rightarrow \lambda^\tau(\mathsf{fchb})$. A translation T between two calculi maps types to types, processes to processes, and contexts to contexts, such that their types correspond. Good translations between two calculi C and C' should have the following useful properties:

Definition 2.2.7 (Full Abstraction,[SSNSS08])**.** Let T be a translation, then T is *fully abstract* if $\forall \ p_1, p_2$:

$$T(p_1) \leq_{C',T(\tau)} T(p_2) \ \Leftrightarrow \ p_1 \leq_{C,\tau} p_2$$

Definition 2.2.8 (Convergence Equivalence,[SSNSS08])**.** Let T be a translation, then T is *convergence equivalent* if $\forall\ p_1, p_2$:

$$T(p)\Downarrow \ \Leftrightarrow\ p\Downarrow \text{ and } T(p)\downarrow \ \Leftrightarrow\ p\downarrow$$

Definition 2.2.9 (Compositionality,[SSNSS08])**.** Let T be a translation, then T is *compositional* if

$$\forall \text{ contexts } D \text{ and processes } p\text{: } T(D)[T(p)] = T(D[p]).$$

Theorem 2 (Adequacy,[SSNSS08])**.** *If a translation T is compositional and convergence equivalent, then T is adequate.*

Summary

In this Section of Chapter 2 we learned about syntax and semantics of the $\lambda^\tau(\mathsf{fchb})$ calculus. We now know about the concept of futures, handles, cells and processes. We read about how types can be constructed in $\lambda^\tau(\mathsf{fchb})$. We can use reduction rules to reduce a term in the *lambda calculus with futures*. Further more we saw important properties of process components: well-formedness, successfulness and observability. The observability lead us to a observational semantic. This semantic supports defining a further property of process components: The may-must-convergence. With this in mind, we are able to proof program equivalences in $\lambda^\tau(\mathsf{fchb})$. Finally we learned about how 'good' translations of λ-calculi can be identified in general.

2.3 Encoding Primitives

In this section we extend the syntax and semantics of $\lambda^\tau(\mathsf{fchb})$ to provide new concurrency primitives. For every new primitive we give an informal description of its behaviour and a formal specification of the primitive followed by a syntactical and semantical extension. The syntactical extension includes new types and typing rules. The semantic extension consists of new reduction rules and new contexts. For each primitive we give a translation to $\lambda^\tau(\mathsf{fchb})$ if neccessary. The translation consists of an implementation of the primitive in terms of the $\lambda^\tau(\mathsf{fchb})$ calculus, a type mapping and new transformation rules.

2.3.1 Test And Set

The *Test and Set* [Tan01] primitive is usually implemented in the hardware of every modern computer system. Test and set models the operations of testing a variable against a condition and setting its value if the test did not fail in one atomic step.

Formal Specification. If the condition is false, the thread sets it to true, enters the critical code and resets the value to false. If the condition is true, the calling thread must wait or do something else until it is set to false.

$$\begin{array}{rl} \mathsf{TSet} :: & \mathsf{ref}\ \alpha \to \beta \to \mathsf{bool} \\ \mathsf{TSet} ::= & \lambda xy.\textbf{case exch}(x, \mathsf{True})\ \textbf{of} \\ & \qquad\qquad \mathsf{True} \to\ \mathsf{False} \\ & \qquad \mathsf{False} \to\ y; \textbf{exch}(x, \mathsf{False}) \end{array}$$

Figure 2.12: Encoding of Test and Set in $\lambda^\tau(\mathsf{fchb})$.

Implementation Of Test And Set. We give an implementation of the test and set in $\lambda^\tau(\mathsf{fchb})$. X represents a cell. Y is the critical code. The $\textbf{exch}(x, 1)$ operation in the **case** expression sets the value of the cell to the blocking state 1. If the exchanged value had already been 1, then the calling thread must wait and may retry accessing X at a later date. Otherwise, the calling thread becomes the blocker itself and is allowed to execute the critical code. For the typing we assume a boolean type as presented in Example 2.2.1. A translation is not neccessary since the implementation of test and set is denoted in the $\lambda^\tau(\mathsf{fchb})$ calculus.

2.3.2 Channels

A channel is a list of elements of type τ with a read-end at one side and a write-end at the other. Elements put into the channel can be read out in a *first in, first out* order. A read and a write operation can be executed in parallel by several threads. A channel has no capacity bounding.

Syntax extension:

$$\begin{array}{rcl}
\tau \in \mathit{Type} & ::= & \mathsf{chan}\ \tau \mid \ldots \\
c \in \mathit{Const} & ::= & \mathbf{newChan}^{\tau} \mid \mathbf{readChan}^{\tau} \mid \ldots \\
e \in \mathit{Exp} & ::= & \mathbf{writeChan}(e_1, e_2) \mid \ldots \\
p \in \mathit{Proc} & ::= & c\,\mathsf{ch}\,[v_1 \ldots v_n] \mid c\,\mathsf{ch}\ - \mid \ldots
\end{array}$$

Extensions of the type system:

$$\begin{array}{rl}
\mathbf{newChan}^{\tau}: & \mathsf{unit} \to \mathsf{chan}\ \tau \\
\mathbf{readChan}^{\tau}\ c: & \mathsf{chan}\ \tau \to \tau \\
\mathbf{writeChan}\ c\ v: & \tau \to \mathsf{chan}\ \tau \to \mathsf{unit}
\end{array}
\qquad
\frac{}{x\,\mathsf{ch} - : wt}
\qquad
\frac{x : \tau \quad v_i : \tau}{x\,\mathsf{ch}\,[v_1, \ldots, v_n] : wt}$$

Extensions of the reduction rules:

$$\begin{array}{ll}
(\textsc{chan.new}(\mathsf{ev})) & E[\mathbf{newChan}^{\tau}\ c] \to (\nu c)(E[c] \mid c\,\mathsf{ch} -) \\
(\textsc{chan.read}(\mathsf{ev})) & E[\mathbf{readChan}^{\tau}\ c] \mid c\,\mathsf{ch}\,[v_1 \ldots v_n] \to E[v_1] \mid c\,\mathsf{ch}\,[v_2, \ldots, v_n] \\
(\textsc{chan.write}(\mathsf{ev})) & E[\mathbf{writeChan}(c, v_{n+1})] \mid c\,\mathsf{ch}\,[v_1 \ldots v_n] \\
 & \to E[\mathsf{unit}] \mid c\,\mathsf{ch}\,[v_1, \ldots, v_n, v_{n+1}] \text{ for } 0 \le n \\
 & E[\mathbf{writeChan}(c, (w_1, \ldots, w_m))] \mid c\,\mathsf{ch}\,[v_1 \ldots v_n] \\
 & \to E[\mathsf{unit}] \mid c\,\mathsf{ch}\,[v_1 \ldots v_n, w_1, \ldots, w_m] \text{ for } 0 \le n, m
\end{array}$$

New contexts:

$$\begin{array}{rcl}
\widetilde{E} & ::= & \mathbf{writeChan}(\widetilde{E}, e) \mid \mathbf{writeChan}(c, \widetilde{E}) \mid \ldots \\
\widetilde{F} & ::= & \widetilde{E}[\mathbf{writeChan}([\,], v)] \mid \widetilde{E}[\mathbf{readChan}^{\tau}\,[\,]] \mid \ldots
\end{array}$$

Figure 2.13: Extensions of $\lambda^{\tau}(\mathsf{fchb})$ for $\lambda^{\tau}(\mathsf{fchb}) + \mathsf{chan}$

Formal Specification. In Fig. 2.13 the syntactical extension from $\lambda^{\tau}(\mathsf{fchb})$ to a calculus $\lambda^{\tau}(\mathsf{fchb}) + \mathsf{chan}$ is shown. A channel has a new type constructor chan of arity 1. Channel components $c\,\mathsf{ch}\,[v_1 \ldots v_n]$ associate channels c to a list of elements $v_1 \ldots v_n$ representing the channel content. $c\,\mathsf{ch}-$ represents an empty channel. There are new constants $\textbf{newChan}^{\tau}$ to spawn a new channel with an empty list and $\textbf{readChan}^{\tau}$ to get the content of a channel. **writeChan** writes a new value into a channel. Placing the new constants into an evaluation context produces four new reduction rules. Note that the reduction rule (CHAN.READ(ev)) can only be applied on a non-empty channel. Therefore a channel implicitly blocks a read operation on the empty channel. If a channel component or a value written to a channel itself are evaluation contexts, then **writeChan** will reduce in a normal call-by-value EC. If the hole marker is at the position of the channel, then the **writeChan** operation reduces in a future context. Any read operation may have lazy computation, so it reduces inside a future context as well.

Implementing Channels Using Buffers.

```
newChan≙λ_. let hole = (newBuf unit),
                read = (newBuf unit),
                write = (newBuf unit)
            in put(read, hole);
               put(write, hole);
               (read, write)

readChan≙λ⟨read, write⟩.
    let r = (get read),
        hd = (get r),
    in case hd of
        Item(res, tl) ⇒ put(read, tl);
                        res

writeChan≙λ⟨read, write⟩, v.
    let newhole = (newBuf unit),
        oldhole = (get write)
    in put(write, newhole);
       put(oldhole, Item(v, newhole))
```

Figure 2.14: Encoding of channels using buffers.

Inspired by `Haskell`'s channel implementation we denote a channel as a linked list of buffers with a read-end and a write-end. This is illustrated in Fig. 2.3.2. Once a read operation is executed on a channel, it returns the value from the read-end's referenced buffer and moves the next buffer to the read-end's pointer. A write operation firstly writes a value to the last buffer in the chain and then appends a new buffer where its write pointer moves. newChan introduces a new channel component and its members which are a read, a write and a hole buffer. writeChan firstly creates a new empty buffer.

Then it reads the write-ends buffer h_n. Secondly it creates a new item and puts the new buffer and the value into it. This item is the content for the buffer h_n. Finally, it writes the new empty buffer to the write-end. readChan can only apply on a non-empty channel. It returns the value of the item's second value in read-end's buffer and writes the item's first value to the read buffer.

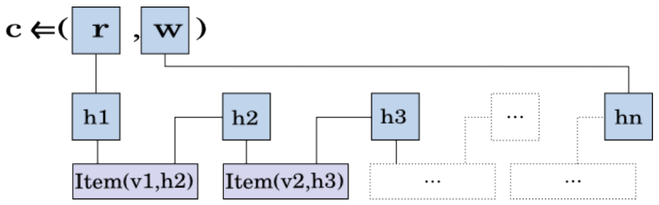

Figure 2.15: A channel with n-1 Items

Note. Later in chapter 3, we will use writeChan$c$ $(v_1 \ldots v_n)$ as new shorthand for writeChan $c\ v_1$; writeChan $c\ v_2; \ldots$; writeChan $c\ v_n$.

The types of the implemented operations are

$$\begin{aligned}
\mathsf{newChan} ::&\ \ \mathsf{unit} \rightarrow (\mathsf{buf}\ (\mathsf{buf}\ (\mathsf{item}\ \tau)), \mathsf{buf}(\mathsf{buf}\ (\mathsf{item}\ \tau)))\\
\mathsf{readChan} ::&\ \ (\mathsf{buf}\ (\mathsf{buf}\ (\mathsf{item}\ \tau)), \mathsf{buf}(\mathsf{buf}\ (\mathsf{item}\ \tau))) \rightarrow \tau\\
\mathsf{writeChan} ::&\ \ (\mathsf{buf}\ (\mathsf{buf}\ (\mathsf{item}\ \tau)), \mathsf{buf}(\mathsf{buf}\ (\mathsf{item}\ \tau))) \rightarrow \tau \rightarrow \mathsf{unit}
\end{aligned}$$

Here, we assume a type constructor item with a data constructor Item. Given this encoding we have new transformation rules in λ^τ(fchb):

(TCHAN.NEW)

$$E[\mathsf{newChan\ unit}]$$
$$\rightarrow (\nu c)(\nu r)(\nu w)(\nu h_1)(E[c] \mid c \Leftarrow (r, w) \mid r\, \mathsf{b}\, h_1 \mid h_1\, \mathsf{b}\ - \mid w\, \mathsf{b}\, h_1)$$

(TCHAN.WRITE) for $n \geq 0$

$$(\nu c)(\nu r)(\nu w)(\nu h_1, \ldots, h_n)$$
$$(E[\mathsf{writeChan}\ (c, v)] \mid c \Leftarrow (r, w) \mid r\, \mathsf{b}\, h_1$$
$$\mid h_1\, \mathsf{b}\, \mathsf{Item}(v_1, h_2) \mid \ \ldots \mid h_{n-1}\, \mathsf{b}\, \mathsf{Item}(v_{n-1}, h_n) \mid h_n\, \mathsf{b}\ - \mid w\, \mathsf{b}\, h_n)$$
$$\rightarrow (\nu c)(\nu r)(\nu w)(\nu h_1, \ldots, h_{n+1})$$
$$(E[\mathsf{unit}] \mid c \Leftarrow (r, w) \mid r\, \mathsf{b}\, h_1 \mid h_1\, \mathsf{b}\, \mathsf{Item}(v_1, h_2) \mid \ \ldots$$
$$\mid h_{n-1}\, \mathsf{b}\, \mathsf{Item}(v_{n-1}, h_n) \mid h_n\, \mathsf{b}\, \mathsf{Item}(v, h_{n+1}) \mid h_{n+1}\, \mathsf{b}\ - \mid w\, \mathsf{b}\, h_{n+1})$$

(TCHAN.WRITEN) for $n, m \geq 0$

$$
\begin{array}{l}
(\nu c)(\nu r)(\nu w)(\nu h_1, \dots, h_n) \\
(E[\mathsf{writeChan}\ (c, (w_1, \dots, w_m))] \mid c \Leftarrow (r, w) \mid r\,\mathsf{b}\,h_1 \\
\mid h_1\,\mathsf{b}\,\mathsf{Item}(v_1, h_2) \mid \dots \mid h_{n-1}\,\mathsf{b}\,\mathsf{Item}(v_{n-1}, h_n) \mid h_n\,\mathsf{b}\,- \mid w\,\mathsf{b}\,h_n) \\
\to (\nu c)(\nu r)(\nu w)(\nu h_1, \dots, h_n, h_{n+1}, \dots, h_{n+m}) \\
(E[\mathsf{unit}] \mid c \Leftarrow (r, w) \mid r\,\mathsf{b}\,h_1 \mid h_1\,\mathsf{b}\,\mathsf{Item}(v_1, h_2) \mid \dots \\
\mid h_{n-1}\,\mathsf{b}\,\mathsf{Item}(v_{n-1}, h_n) \mid h_n\,\mathsf{b}\,\mathsf{Item}(w_1, h_{n+1}) \mid \dots \\
\mid h_{n+m-1}\,\mathsf{b}\,\mathsf{Item}(w_m, h_{n+m}) \mid h_{n+m}\,\mathsf{b}\,- \mid w\,\mathsf{b}\,h_{n+m})
\end{array}
$$

(TCHAN.READ) for $n > 0$

$$
\begin{array}{l}
(\nu c)(\nu r)(\nu w)(\nu h_1, \dots, h_n) \\
(E[\mathsf{readChan}\ c] \mid c \Leftarrow (r, w) \mid r\,\mathsf{b}\,h_1 \\
\mid h_1\,\mathsf{b}\,\mathsf{Item}(v_1, h_2) \mid \dots \mid h_{n-1}\,\mathsf{b}\,\mathsf{Item}(v_{n-1}, h_n) \mid h_n\,\mathsf{b}\,- \mid w\,\mathsf{b}\,h_n) \\
\to (\nu c)(\nu r)(\nu w)(\nu h_2, \dots, h_n) \\
(E[v_1] \mid c \Leftarrow (r, w) \mid r\,\mathsf{b}\,h_2 \mid h_2\,\mathsf{b}\,\mathsf{Item}(v_2, h_3) \mid \dots \\
\mid h_{n-1}\,\mathsf{b}\,\mathsf{Item}(v_{n-1}, h_n) \mid h_n\,\mathsf{b}\,- \mid w\,\mathsf{b}\,h_n
\end{array}
$$

In chapter 3 we will show that these transformation rules are correct.

Translation From λ^τ(fchb) + chan $\to$ λ^τ(fchb).

$$
\begin{array}{rcl}
T(c\,\mathsf{ch}\,[v_0, \dots, v_{n-1}]) & \mapsto & c \Leftarrow (r, w) \mid r\,\mathsf{b}\,h_0 \mid \\
 & & h_0\,\mathsf{b}\,\mathsf{Item}(v_0, h_1) \mid \dots \mid h_{n-1}\,\mathsf{b}\,\mathsf{Item}(v_{n-1}, h_n) \mid \\
 & & h_n\,\mathsf{b}\,- \mid w\,\mathsf{b}\,h_n \mid \dots \\
T(c\,\mathsf{ch}\,-) & \mapsto & c \Leftarrow (r, w) \mid r\,\mathsf{b}\,h_0 \mid h_0\,\mathsf{b}\,- \mid w\,\mathsf{b}\,h_0
\end{array}
$$

Figure 2.16: Translation of components from λ^τ(fchb) + chan to λ^τ(fchb)

The implementation of channels with buffers gives rise to a translation T from λ^τ(fchb) + chan $\to$ λ^τ(fchb). The translation maps constants in λ^τ(fchb) + chan to the implementation in λ^τ(fchb): $T(\textbf{newChan}^\tau) \mapsto \mathsf{newChan}$, $T(\textbf{readChan}^\tau) \mapsto \mathsf{readChan}$ and $T(\textbf{writeChan}) \mapsto \mathsf{writeChan}$. In Fig. 2.16 a translation for the process components is shown. The channel type translation is given by $T(\mathsf{chan}\ \tau) \mapsto (\mathsf{buf}\ (\mathsf{buf}\ (\mathsf{item}\ T(\tau))), \mathsf{buf}(\mathsf{buf}\ (\mathsf{item}\ T(\tau))))$. Besides this, the notion of successful processes in λ^τ(fchb) is extended by components $y\,\mathsf{ch}\,[v_1 \dots v_n]$ and $y\,\mathsf{ch}\,-$ as successful processes. These processes are well-formed if they do not introduce a variable twice.

Note. Giving evidence for this translation is not part of this thesis, since this task exceeds the scope of a bachelor's thesis.

Additional Channel Operations. We extend the set of operations on

```
mergeList2Chan $\hat{=}\lambda l_1\ l_2.$
  let $c_m$ = (newChan unit)
  in thread($\lambda\_.$merge $l_1\ c_m$);
     thread($\lambda\_.$merge $l_2\ c_m$); $c_m$

merge $\hat{=}\lambda l\ c.$case $l$ of
  $x : xs \rightarrow$ writeChan $c\ x$
                   merge $xs\ c$
  $x$      $\rightarrow$ writeChan $c\ x$
```

Figure 2.17: A merge operation of two lists to one channel.

channels by a merge operation (Fig. 2.3.2). mergeList2Chan takes two lists and merges them to a channel with a recursive function merge using two threads in parallel. After finishing merging it returns the new channel.

A more tricky version could be a merge of two channels to one channel, Fig. 2.3.2. The operation mergeChan2Chan takes two channels and merges them to a new channel. But there is a limitation in use of this implementation: Because we have no chance to determine when a channel is empty, we need a special symbol ϵ which marks a sequence's end. With that the termination of merge$_{c2c}$ is ensured.

```
mergeChan2Chan $\hat{=}\lambda c_1\ c_2.$
  let $c_m$ = (newChan unit)
  in thread($\lambda\_.$merge$_{c2c}$ $c_1\ c_m$);
     thread($\lambda\_.$merge$_{c2c}$ $c_2\ c_m$); $c_m$

merge$_{c2c}$ $\hat{=}\lambda c\ c_m.$
  let $v$ = writeChan $c$
  in case $(v == \epsilon)$ of
    False $\rightarrow$ writeChan $c_m\ v$;
        merge$_{c2c}$ $c\ c_m$
    True $\rightarrow$ unit
```

Figure 2.18: A merge operation of two channels to one channel.

2.3.3 Quantity Semaphores

Syntax extension:

$$\begin{array}{rcl} \tau \in \mathit{Type} & ::= & \mathsf{qsem}\ \tau \mid \ldots \\ c \in \mathit{Const} & ::= & \mathbf{newQSem}^\tau \mid \mathbf{up}^\tau \mid \mathbf{down}^\tau \mid \ldots \\ p \in \mathit{Proc} & ::= & q\,\mathsf{s}_\kappa\,n, (\kappa \in N_0\ \mathit{and}\ 0 \leq n \leq \kappa) \mid \ldots \end{array}$$

Extensions of the type system:

$$\begin{array}{rl} \mathbf{newQSem}^\tau\ \kappa : & \alpha \rightarrow \mathsf{qsem}\ \tau \\ \mathbf{up}^\tau\ q : & \mathsf{qsem}\ \tau \rightarrow \mathsf{unit} \\ \mathbf{down}^\tau\ q : & \mathsf{qsem}\ \tau \rightarrow \mathsf{bool} \end{array} \qquad \frac{q : \mathsf{qsem}\ \tau \quad n, \kappa : \mathsf{Int}}{q\,\mathsf{s}_\kappa\,n : \mathit{wt}}$$

Extensions of the reduction rules:

$$\begin{array}{ll} (\text{QSEM.NEW}(\mathsf{ev})) & E[\mathbf{newQSem}^\tau\ \kappa] \rightarrow (\nu q)(E[q] \mid q\,\mathsf{s}_\kappa\,\kappa) \\ (\text{QSEM.UP}(\mathsf{ev})) & E[\mathbf{up}^\tau\ q] \mid q\,\mathsf{s}_\kappa\,n \rightarrow E[\mathsf{unit}] \mid q\,\mathsf{s}_\kappa\,n+1 \text{ for } n < \kappa \\ (\text{QSEM.DOWN}(\mathsf{ev})) & E[\mathbf{down}^\tau\ q] \mid q\,\mathsf{s}_\kappa\,n \rightarrow E[\mathsf{True}] \mid q\,\mathsf{s}_\kappa\,n-1 \\ & \text{for } 0 < n \leq \kappa \end{array}$$

New contexts:

$$\widetilde{F} \ ::= \ \widetilde{E}[\mathbf{down}^\tau, [\,]] \mid \widetilde{E}[\mathbf{up}^\tau, [\,]] \mid \ldots$$

Figure 2.19: Extensions of $\lambda^\tau(\mathsf{fchb})$ for $\lambda^\tau(\mathsf{fchb}) + \mathsf{qsem}$

A quantity semaphore is initialised with a maximum number of threads that are allowed to run code protected by the semaphore. The value of the semaphore counts the number of additional threads that may execute. If a thread tries to execute the protected code the semaphore's counter decrements. If a thread leaves, the counter increments. If the counter is zero, the quantity semaphore blocks until one of the threads inside exits.

Formal Specification. Fig. 2.19 shows the extension for $\lambda^\tau(\mathsf{fchb})$ to $\lambda^\tau(\mathsf{fchb}) + \mathsf{qsem}$. There is a new type constructor $\mathsf{qsem}\ \tau$. We also have a new process component $q\,\mathsf{s}_\kappa\,n$ where $\kappa \in N_0$ and $n \leq \kappa$. κ denotes the capacity of the quantity semaphore and n counts how many threads may enter additionally. $\mathbf{newQSem}^\tau$ creates a new quantity semaphore construct. $\mathbf{up}^\tau$ increments the count variable if the list is empty. Otherwise it gives signal to a waiting thread from the list. $\textbf{down}^{\tau}$[1] decrements the count variable if it is not *zero* otherwise the calling thread is delayed. The reduction rule ($\text{QSEM.UP}(\mathsf{ev})$) can only apply on quantity semaphore components where the counter value n has not reached the value κ ($n < \kappa$ must hold). In contrast, the reduction rule ($\text{QSEM.DOWN}(\mathsf{ev})$) is not allowed to apply in cases of $n < 0$. Therefore a quantity semaphore implicitly blocks if κ threads have executed $\textbf{down}^{\tau}$.

Implementation Of Quantity Semaphores.

```
newQSem≙λκ.let l = (Nil)
               q = newBuf unit
           in put (q, (κ, l)); q
```

```
up≙λqs.
  let ⟨cnt, ls⟩ = get qs in
  case ls of
  [ ] → put qs (cnt + 1, ls)
  x : xs →  x True;
            put qs (cnt, xs)
```

```
down≙λqs.
  let ⟨cnt, ls⟩ = get qs
  in case (cnt == 0) of
    True →  let ⟨x, f⟩ = newhandled
            in put qs (cnt, x : ls); case x of
              True →  True
              False →  ⊥
    False →  put qs (cnt − 1, ls); True
```

Figure 2.20: Encoding of quantity semaphores using handles.

In terms of the $\lambda^{\tau}(\mathsf{fchb})$ calculus, a quantity semaphore is a buffer holding a tuple of the count variable and a list of handles (Fig. 2.20) (respectively buffers Fig. 2.21) for the waiting threads. newQSem creates a new quantity semaphore as a buffer containing the count variable and an empty list. The operation up increments the count variable if the list is empty. Otherwise it gives signal to a waiting thread by taking the first item from the list and consuming it. In the case of handles, it binds True to the handle. In the case of buffers it puts a value into the buffer. down creates a new handle (resp. a new buffer) that blocks the accessing thread and adds it to the list if the count is *zero*. Otherwise it decrements the count.

[1]In other literature, up and down are often called signal and wait. This describes what they are often used for.

```
up$\hat{=}\lambda qs.$                                   down$\hat{=}\lambda qs.$
    let $\langle cnt, ls\rangle$ = **get** $qs$ in              let $\langle cnt, ls\rangle$ = **get** $qs$
    **case** $ls$ **of**                                 in **case** $(cnt == 0)$ **of**
    $[\,] \rightarrow$ **put** $qs\ (cnt+1, ls)$                 True $\rightarrow$ let $x$ = (**newBuf** unit)
    $x : xs \rightarrow$ **put** $(x,$ True$)$;                           in **put** $qs\ (cnt, x : ls)$; **get** $x$
                 **put** $qs\ (cnt, xs)$                 False $\rightarrow$ **put** $qs\ (cnt-1, ls)$; True
```

Figure 2.21: Encoding of quantity semaphores using buffers.

The types of the implemented operations are

$$\begin{aligned}\mathsf{newQSem} &:: \mathsf{Int} \rightarrow \mathsf{List\ bool} \rightarrow \mathsf{buf\ (Int, [bool])}\\ \mathsf{up} &:: \mathsf{buf\ (Int, [bool])} \rightarrow \mathsf{unit}\\ \mathsf{down} &:: \mathsf{buf\ (Int, [bool])} \rightarrow \mathsf{bool}\end{aligned}$$

Translation From $\lambda^\tau(\mathbf{fchb}) + \mathbf{qsem} \rightarrow \lambda^\tau(\mathbf{fchb})$.

$$T(q\,\mathsf{s}_\kappa\, n) \quad \mapsto \quad q\,\mathsf{b}\,(n, [\,])\ for\ 0 \leq n \leq \kappa$$

Figure 2.22: Translation of components from $\lambda^\tau(\mathsf{fchb}) + \mathsf{qsem}$ to $\lambda^\tau(\mathsf{fchb})$

A translation T from $\lambda^\tau(\mathsf{fchb}) + \mathsf{qsem} \rightarrow \lambda^\tau(\mathsf{fchb})$ maps: $T(\mathbf{newQSem}^\tau) \mapsto \mathsf{newQSem}, T(\mathbf{up}^\tau) \mapsto \mathsf{up}$ and $T(\mathbf{down}^\tau) \mapsto \mathsf{down}$. We give only a translation in terms of handles, since a translation from handles to buffers had been shown in [SSNSS09]. In Fig. 2.22 a translation for the process component is shown. A quantity semaphore contains a tuple of the counter and in the case of $0 < n \leq \kappa$ an empty list. If the counter value is *zero* then the list could be empty or contain handles (resp. buffers) if there are threads blocking on the semaphore. There are new transformation rules for the implemented operations in $\lambda^\tau(\mathsf{fchb})$:

(TQSEM.NEW)
$$E[\mathsf{newQSem}\ \kappa] \rightarrow (\nu q)(E[q] \mid q\,\mathsf{b}\,(\kappa, [\,]))$$

(TUP) for $n < \kappa$ (and an empty list)
$$(\nu q)(E[\mathsf{up}\ q] \mid q\,\mathsf{b}\,(n, [\,])) \rightarrow (\nu q)(E[\mathsf{unit}] \mid q\,\mathsf{b}\,(n+1, [\,]))$$

(TUP) for $n = \kappa$ (and the list may not be empty)
$$(\nu q)(\nu h_1, \ldots, h_m)(\nu f_1, \ldots, f_m)$$

$(E[\mathsf{up}\ q] \mid q\,\mathsf{b}\,(\kappa, [h_1, \ldots, h_m]) \mid h_1\,\mathsf{h}\,f_1 \ldots h_m\,\mathsf{h}\,f_m)$
$\to (\nu q)(\nu h_1, \ldots, h_m)(\nu f_1, \ldots, f_m)$
$(E[\mathsf{unit}] \mid q\,\mathsf{b}\,(\kappa, [h_2, \ldots, h_m]) \mid h_2\,\mathsf{h}\,f_2 \ldots h_m\,\mathsf{h}\,f_m \mid h_1\,\mathsf{h}\,\bullet \mid f_1 \Leftarrow \mathsf{True})$

(TDOWN) for $0 < n \leq \kappa$ (and an empty list):
$(\nu q)(E[\mathsf{down}\ q] \mid q\,\mathsf{b}\,(n, [\,])) \to (\nu q)(E[\mathsf{True}] \mid q\,\mathsf{b}\,(n-1, [\,]))$

(TDOWN) for $n = 0, m \geq 0$:
$(\nu q)(\nu h_1, \ldots, h_m)(\nu f_1, \ldots, f_m)$
$(E[\mathsf{down}\ q] \mid q\,\mathsf{b}\,(0, [h_1, \ldots, h_m]) \mid h_1\,\mathsf{h}\,f_1 \ldots h_m\,\mathsf{h}\,f_m)$
$\to (\nu q)(\nu h)(\nu f)(\nu h_1, \ldots, h_m)(\nu f_1, \ldots, f_m)$
$(E[h] \mid q\,\mathsf{b}\,(0, [h : h_1, \ldots, h_m]) \mid h_1\,\mathsf{h}\,f_1 \ldots h_m\,\mathsf{h}\,f_m \mid h\,\mathsf{h}\,f)$

The rule (TQSEM.NEW) creates a new semaphore with capacity κ stored by the index parameter containing a tuple of κ and an empty list. The list models a waiting queue for blocked threads. Therefore, the rule (TDOWN) decrements n, if n threads may enter ($0 < n \leq \kappa$). In the case of $n = 0$ the context evaluates to a new handle that is appended to the quantity semaphore's waiting queue. The rule (TUP) increments the quantity semaphore's counter value if the queue is empty. Otherwise the counter value remains κ while one of the handles from the list is bound to True and its thread may continue. A mapping of the type qsem τ is given by $T(\mathsf{qsem}\ \tau) : \mathsf{buf}\ (\mathsf{Int}, [\mathsf{bool}])$.

Remark. To ensure that down is always run before up, we could define a function like $\mathsf{enterQSem} \hat{=} \lambda\ q\ c.\mathsf{down}\ q; c; \mathsf{up}\ q$, with q being the semaphore and c the code encapsulated by q.

2.3.4 Bounded Channels

Syntax extension:

$$\begin{array}{rcl}
\tau \in \mathit{Type} & ::= & \mathsf{bchan}\ \tau \mid \dots \\
c \in \mathit{Const} & ::= & \textbf{newBChan}^{\tau} \mid \textbf{readBChan}^{\tau} \mid \dots \\
e \in \mathit{Exp} & ::= & \textbf{writeBChan}(e_1, e_2) \mid \dots \\
p \in \mathit{Proc} & ::= & c\,\mathsf{bch}_{\kappa}\,[v_1, \dots, v_{\kappa}], \kappa \in N_0 \mid c\,\mathsf{bch}_{\kappa}\ - \ \mid \dots
\end{array}$$

Extensions of the type system:

$$\begin{array}{rl}
\textbf{newBChan}^{\tau}\ \kappa : & \alpha \to \mathsf{bchan}\ \tau \\
\textbf{readBChan}^{\tau}\ c : & \mathsf{bchan}\ \tau \to \tau \\
\textbf{writeBChan}\ c\ v : & \mathsf{bchan}\ \tau \to \tau \to \mathsf{unit}
\end{array}
\qquad
\frac{}{c\,\mathsf{bch}_{\kappa} - : \mathit{wt}}
\qquad
\frac{c : \mathsf{bchan}\tau \quad v_i : \tau}{c\,\mathsf{bch}_{\kappa}\,[v_1, \dots, v_{\kappa}] : \mathit{wt}}$$

Extensions of the reduction rules:

$$\begin{array}{ll}
(\text{BCHAN.NEW}(\mathsf{ev})) & E[\textbf{newBChan}^{\tau}\ \kappa], \kappa \in N_0 \to (\nu c)(E[c] \mid c\,\mathsf{bch}_{\kappa} -) \\
(\text{BCHAN.READ}(\mathsf{ev})) & E[\textbf{readBChan}^{\tau}\ c] \mid c\,\mathsf{bch}_{\kappa}\,[v_1, \dots, v_{\kappa}] \\
 & \to E[v_1] \mid c\,\mathsf{bch}_{\kappa}\,[v_2, \dots, v_{\kappa}] \\
(\text{BCHAN.WRITE}(\mathsf{ev})) & E[\textbf{writeBChan}(c, v)] \mid c\,\mathsf{bch}_{\kappa} - \to E[\mathsf{unit}] \mid c\,\mathsf{bch}_{\kappa}\,[v] \\
 & E[\textbf{writeBChan}(c, v_{n+1})] \mid c\,\mathsf{bch}_{\kappa}\,[v_1 \dots v_n] \\
 & \to E[\mathsf{unit}] \mid c\,\mathsf{bch}_{\kappa}\,[v_1 \dots v_n, v_{n+1}] \text{ for } n < \kappa
\end{array}$$

New contexts:

$$\begin{array}{rcl}
\widetilde{E} & ::= & \textbf{writeBChan}(\widetilde{E}, e) \mid \textbf{writeBChan}(c, \widetilde{E}) \\
\widetilde{F} & ::= & \widetilde{E}[\textbf{writeBChan}([\,], v)] \mid \widetilde{E}[\textbf{readBChan}^{\tau}\,[\,]]
\end{array}$$

Figure 2.23: Extensions of $\lambda^{\tau}(\mathsf{fchb})$ for $\lambda^{\tau}(\mathsf{fchb}) + \mathsf{bchan}$

A bounded channel is a channel that has a limited capacity. Once the capacity is reached, it blocks the write operation until a place gets free because a read operation was performed.

Formal Specification. There is a new type constructor $\mathsf{bchan}\ \tau$ for the new channel type. Fig. 2.23 shows operations on bounded channels similar to the ones for ordinary channels. $\textbf{newBChan}^{\tau}$ spawns a new bounded channel with a given capacity κ. The constant $\textbf{readBChan}^{\tau}$ reads a value from the channel and decrements the current capacity count. If the channel is empty, the calling thread suspends while no value is available. **writeBChan** writes a value to the channel and increments the current capacity count. If the current capacity count has reached the maximum capacity the thread suspends on the computation until a place becomes free. A bounded channel process component $c\,\mathsf{bch}_\kappa\,[v_1, \ldots, v_\kappa]$ consists of the capacity $\kappa \in N_0$ and a list of values. The empty bounded channel component is denoted as $c\,\mathsf{bch}_\kappa\,-$. A bounded channel's reduction rules are similar to the channel ones: There is only an additional restriction for the write operation that is only allowed to be performed when the capacity κ has not been reached.

Implementation Of Bounded Channels.

```
newBChan≙λκ.
   let qsem = (newQSem κ);
       chan = (newChan unit)
   in (chan, qsem)
```

```
readBChan≙λ⟨c, qsem⟩.
    up qsem;
    readChan c
```

```
writeBChan≙λ⟨c, qsem⟩, v_.
    down qsem;
    writeChan c v
```

Figure 2.24: Encoding of bounded channels using a channel and a qsem.

Using a quantity semaphore with buffers or handles, the implementation of bounded channels is straightforward. A bounded channel is implemented as a tuple of a quantity semaphore and a channel. For each entry in the bounded channel the semaphore decrements until it reaches *zero*. Then threads writing to a bounded channel get suspended on the quantity semaphore. In a read operation the quantity semaphore's counter increments and the value of the channel's read-end is returned. In this case a suspended writing thread my continue. The types are:

$$\begin{aligned}
\mathsf{newBChan} ::\ & \mathsf{ref}\ \alpha \to (\mathsf{ref}\ \alpha, \mathsf{List\ bool}) \to (\mathsf{buf}\ (\mathsf{buf}\ (\mathsf{item}\ \tau)), \mathsf{buf}(\mathsf{buf}\ (\mathsf{item}\ \tau))) \\
& \to ((\mathsf{buf}\ (\mathsf{buf}\ (\mathsf{item}\ \tau)), \mathsf{buf}(\mathsf{buf}\ (\mathsf{item}\ \tau))), (\mathsf{ref}\ \alpha, \mathsf{List\ bool})) \\
\mathsf{readBChan} ::\ & ((\mathsf{buf}\ (\mathsf{buf}\ (\mathsf{item}\ \tau)), \mathsf{buf}(\mathsf{buf}\ (\mathsf{item}\ \tau))), (\mathsf{ref}\ \alpha, \mathsf{List\ bool})) \to \tau \\
\mathsf{writeBChan} ::\ & \tau \to ((\mathsf{buf}\ (\mathsf{buf}\ (\mathsf{item}\ \tau)), \mathsf{buf}(\mathsf{buf}\ (\mathsf{item}\ \tau))), (\mathsf{ref}\ \alpha, \mathsf{List\ bool})) \to \mathsf{unit}
\end{aligned}$$

Translation From $\lambda^\tau(\textbf{fchb}) + \textbf{bchan} \to \lambda^\tau(\textbf{fchb})$.

$$\begin{array}{rcl} T(c\,\mathsf{bch}_\kappa\,-) & \mapsto & c \Leftarrow (c', q) \mid q\,\mathsf{b}\,(\kappa, [\,]) \mid c' \Leftarrow (r, w) \mid r\,\mathsf{b}\,b_1 \mid \\ & & b_1\,\mathsf{b}\,- \mid w\,\mathsf{b}\,b_1 \\ T(c\,\mathsf{bch}_\kappa\,[v_1, \ldots, v_n]) & \mapsto & c \Leftarrow (c', q) \mid q\,\mathsf{b}\,(\kappa - n, [\,]) \mid c' \Leftarrow (r, w) \mid r\,\mathsf{b}\,b_1 \mid \\ & & b_1\,\mathsf{b}\,\mathsf{Item}(v_1, b_1) \mid \ldots \mid b_n\,\mathsf{b}\,\mathsf{Item}(v_n, b_{n+1}) \mid \\ & & b_{n+1}\,\mathsf{b}\,- \mid w\,\mathsf{b}\,b_{n+1}\ \textit{for}\ 0 < n \leq \kappa \end{array}$$

Figure 2.25: Translation of components from $\lambda^\tau(\mathsf{fchb}) + \mathsf{bchan}$ to $\lambda^\tau(\mathsf{fchb})$

A translation T maps **writeBChan** $\to$ **writeBChan**, **readBChan**$^\tau$ $\to$ readBChan and **newBChan**$^\tau$ $\to$ newChan. In Fig. 2.25 a bounded channel component is translated to a tuple containing a non-bounded channel and a quantity semaphore that counts the entries of the channel.

(TBCHAN.NEW)

$$\begin{array}{l} E[\mathsf{newBChan}\ \kappa] \\ \to (\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_1) \\ (E[c] \mid c \Leftarrow (c', q) \mid q\,\mathsf{b}\,(\kappa, [\,]) \mid c' \Leftarrow (r, w) \mid r\,\mathsf{b}\,b_1 \mid b_1\,\mathsf{b}\,- \mid w\,\mathsf{b}\,b_1) \end{array}$$

(TBCHAN.WRITE) for $0 < n \leq \kappa$

$$\begin{array}{l} (\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_1 \ldots b_{n+1}) \\ (E[\mathsf{writeBChan}\ (c, v)] \mid c \Leftarrow (c', q) \mid q\,\mathsf{b}\,(\kappa - n, [\,]) \mid c' \Leftarrow (r, w) \\ \mid r\,\mathsf{b}\,b_1 \mid b_1\,\mathsf{b}\,\mathsf{Item}(v_1, b_2) \mid \ldots \mid b_n\,\mathsf{b}\,\mathsf{Item}(v_n, b_{n+1}) \mid b_{n+1}\,\mathsf{b}\,- \\ \mid w\,\mathsf{b}\,b_{n+1}) \\ \to (\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_1 \ldots b_{n+2}) \\ (E[\mathsf{unit}] \mid c \Leftarrow (c', q) \mid q\,\mathsf{b}\,(\kappa - n - 1, [\,]) \mid c' \Leftarrow (r, w) \\ \mid r\,\mathsf{b}\,b_1 \mid b_1\,\mathsf{b}\,\mathsf{Item}(v_1, b_2) \mid \ldots \mid b_n\,\mathsf{b}\,\mathsf{Item}(v_n, b_{n+1}) \mid b_{n+1}\,\mathsf{b}\,\mathsf{Item}(v, b_{n+2}) \\ \mid b_{n+2}\,\mathsf{b}\,- \mid w\,\mathsf{b}\,b_{n+2}) \end{array}$$

(TBCHAN.WRITE) for $n > \kappa$

$$\begin{array}{l} (\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_1 \ldots b_{\kappa+1})(\nu h_1, \ldots, h_m)(\nu f_1, \ldots, f_m) \\ (E[\mathsf{writeBChan}\ (c, v)] \mid c \Leftarrow (c', q) \mid q\,\mathsf{b}\,(0, [h_1, \ldots, h_m]) \mid c' \Leftarrow (r, w) \\ \mid r\,\mathsf{b}\,b_1 \mid b_1\,\mathsf{b}\,\mathsf{Item}(v_1, b_2) \mid \ldots \mid b_\kappa\,\mathsf{b}\,\mathsf{Item}(v_\kappa, b_{\kappa+1}) \mid b_{\kappa+1}\,\mathsf{b}\,- \\ \mid w\,\mathsf{b}\,b_{\kappa+1} \mid h_1\,\mathsf{h}\,f_1 \mid \ldots \mid h_m\,\mathsf{h}\,f_m) \\ \to (\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_1 \ldots b_{\kappa+1})(\nu h_1, \ldots, h_{m+1})(\nu f_1, \ldots, f_{m+1}) \\ (E[\mathsf{wait}\ h_{m+1}] \mid c \Leftarrow (c', q) \mid q\,\mathsf{b}\,(0, [h_1, \ldots, h_{m+1}]) \mid c' \Leftarrow (r, w) \end{array}$$

$\mid r \,\mathsf{b}\, b_1 \mid b_1 \,\mathsf{b}\, \mathsf{Item}(v_1, b_2) \mid \ldots \mid b_\kappa \,\mathsf{b}\, \mathsf{Item}(v_\kappa, b_{\kappa+1}) \mid b_{\kappa+1} \,\mathsf{b}\, -$
$\mid w \,\mathsf{b}\, b_{\kappa+1} \mid h_1 \,\mathsf{h}\, f_1 \mid \ldots \mid h_{m+1} \,\mathsf{h}\, f_{m+1})$

(TBCHAN.READ) for $0 < n \leq \kappa$
$(\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_1 \ldots b_{n+1})$
$(E[\mathsf{readBChan}\ c] \mid c \Leftarrow (c', q) \mid q \,\mathsf{b}\, (\kappa - n, [\,]) \mid c' \Leftarrow (r, w) \mid r \,\mathsf{b}\, b_1$
$\mid b_1 \,\mathsf{b}\, \mathsf{Item}(v_1, b_2) \mid \ldots \mid b_n \,\mathsf{b}\, \mathsf{Item}(v_n, b_{n+1}) \mid b_{n+1} \,\mathsf{b}\, - \mid w \,\mathsf{b}\, b_{n+1})$
$\rightarrow (\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_2 \ldots b_{n+1})$
$(E[v_1] \mid c \Leftarrow (c', q) \mid q \,\mathsf{b}\, (\kappa - n + 1, [\,]) \mid c' \Leftarrow (r, w)$
$\mid r \,\mathsf{b}\, b_2 \mid b_2 \,\mathsf{b}\, \mathsf{Item}(v_2, b_3) \mid \ldots \mid b_n \,\mathsf{b}\, \mathsf{Item}(v_n, b_{n+1}) \mid b_{n+1} \,\mathsf{b}\, - \mid w \,\mathsf{b}\, b_{n+1})$

(TBCHAN.READ) for $n > \kappa$
$(\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_1 \ldots b_{\kappa+1})(\nu h_1 \ldots h_m)(\nu f_1 \ldots f_m)$
$(E[\mathsf{readBChan}\ c] \mid c \Leftarrow (c', q) \mid q \,\mathsf{b}\, (0, [h_1, \ldots, h_m]) \mid c' \Leftarrow (r, w) \mid r \,\mathsf{b}\, b_1$
$\mid b_1 \,\mathsf{b}\, \mathsf{Item}(v_1, b_2) \mid \ldots \mid b_\kappa \,\mathsf{b}\, \mathsf{Item}(v_\kappa, b_{\kappa+1}) \mid b_{\kappa+1} \,\mathsf{b}\, -$
$\mid w \,\mathsf{b}\, b_{\kappa+1} \mid h_1 \,\mathsf{h}\, f_1 \mid \ldots \mid h_m \,\mathsf{h}\, f_m)$
$\rightarrow (\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_2 \ldots b_{n+1})(\nu h_2 \ldots h_m)(\nu f_1 \ldots f_m)$
$(E[v_1] \mid c \Leftarrow (c', q) \mid q \,\mathsf{b}\, (0, [h_2, \ldots, h_m]) \mid c' \Leftarrow (r, w) \mid r \,\mathsf{b}\, b_2$
$\mid b_2 \,\mathsf{b}\, \mathsf{Item}(v_2, b_3) \mid \ldots \mid b_\kappa \,\mathsf{b}\, \mathsf{Item}(v_\kappa, b_{\kappa+1}) \mid b_{\kappa+1} \,\mathsf{b}\, -$
$\mid w \,\mathsf{b}\, b_{\kappa+1} \mid h_1 \,\mathsf{h}\, \bullet \mid f_1 \Leftarrow \mathsf{True} \mid h_2 \,\mathsf{h}\, f_2 \mid \ldots \mid h_m \,\mathsf{h}\, f_m)$

The new transformation rules are similar to the channel ones combined with the rules for quantity semaphores. (TBCHAN.NEW) creates the new bounded channel. The rule (TBCHAN.WRITE) evaluates to unit until the capacity of κ reached. If the channel contains already κ entries, then (TBCHAN.WRITE) evaluates to the handle created by the quantity semaphore that blocks the write operation. (TBCHAN.READ) is only defined on a non-empty bounded channel. The bounded channel type iy mapped via $T(\mathsf{bchan})$ to
$((\mathsf{buf}\ (\mathsf{buf}\ (\mathsf{item}\ T(\tau))), \mathsf{buf}(\mathsf{buf}\ (\mathsf{item}\ T(\tau)))), (\mathsf{ref}\ T(\alpha), \mathsf{List\ bool}))$.

2.3.5 Rendezvous And Barrier

Syntax extension:

$$\begin{array}{rcl} \tau \in \mathit{Type} & ::= & \mathsf{bar}\ \tau \mid \ldots \\ c \in \mathit{Const} & ::= & \mathbf{newBar}^{\tau} \mid \mathbf{syncBar}^{\tau} \mid \ldots \\ p \in \mathit{Proc} & ::= & x\ \mathsf{bar}\ \kappa \end{array}$$

Extensions of the type system:

$$\begin{array}{rcl} \mathbf{newBar}^{\tau} & : & \mathsf{Int} \rightarrow \mathsf{bar}\ \tau \\ \mathbf{syncBar}^{\tau}\ x : & & \mathsf{bar}\ \tau \rightarrow (\mathsf{bar}\ \tau \rightarrow \mathsf{bool}) \rightarrow \mathsf{unit} \end{array} \qquad \frac{x : \tau}{x\ \mathsf{bar}\ \kappa : \mathit{wt}}$$

Extensions of the reduction rules:

$$\begin{array}{ll} (\textsc{bar.new}(\mathsf{ev})) & E[\mathbf{newBar}^{\tau}\ \kappa], \kappa\ \in N_0 \rightarrow (\nu x)(E[x] \mid x\ \mathsf{bar}\ \kappa) \\ (\textsc{bar.sync}(\mathsf{ev})) & (\nu x)(E_1[\mathbf{syncBar}^{\tau}\ x] \mid \ldots \mid E_{\kappa}[\mathbf{syncBar}^{\tau}\ x] \mid x\ \mathsf{bar}\ \kappa) \\ & \rightarrow (\nu x)(E_1[\mathsf{True}] \mid \ldots \mid E_{\kappa}[\mathsf{True}] \mid x\ \mathsf{bar}\ \kappa) \end{array}$$

New contexts:

$$\widetilde{E} \ ::= \ \widetilde{E}[\mathbf{syncBar}^{\tau}\ x] \mid \ldots$$

Figure 2.26: Extensions of $\lambda^{\tau}(\mathsf{fchb})$ for $\lambda^{\tau}(\mathsf{fchb}) + \mathsf{bar}$

A rendezvous ensures that two threads meet at a specific point before continuing their computation. The rendezvous idiom blocks at this point until both threads have arrived. A barrier (Fig. 2.27) is a rendezvous for a group of processes. Assume that a application is divided into phases where a couple of threads compute several interims to be the input for the next phase. Then all threads must complete the current phase before entering the next. To achieve this behaviour a barrier is placed at the end of a phase. Note that a barriers is purely for synchronisation and not for exchange of data.

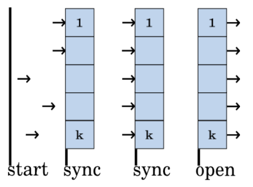

Figure 2.27: The barrier specification

Formal Specification. A barrier component $x\,\mathsf{bar}\,\kappa$ of type $\mathsf{bar}\ \tau$ with capacity κ will be created by $\textbf{newBar}^{\tau}\ \kappa$. It stores a variable κ that stands for the number of threads that may synchronise on the barrier. Threads that have synchronised on it via $\textbf{syncBar}^{\tau}$ will be blocked until all other threads reach the same evaluation context. Therefore the reduction rule (BAR.SYNC(ev)) is only applicable on programs with κ evaluation contexts $E[\textbf{syncBar}^{\tau}]$. In this case the rule forces all contexts to reduce to True.

Implementing Barriers.

```
newBar$\hat{=}\lambda\kappa$.
  let $cnt_{act}$ = (newBuf unit),
      $cnt_{fin}$ = (newBuf unit)
  in put($cnt_{act}, \kappa$);
      put($cnt_{fin}$, Nil);
      ($cnt_{act}, cnt_{fin}, \kappa$)

openBar$\hat{=}\lambda l$.case $l$ of
  ($x : xs$) $\rightarrow$ $x$ True;
                openBar $xs$
  [ ] $\rightarrow$ unit

syncBar$\hat{=}\lambda\langle cnt_{act}, cnt_{fin}, \kappa\rangle$.
  let $act$ = (get $cnt_{act}$),
      $fin$ = (get $cnt_{fin}$)
  in case ($act == 1$) of
      True $\rightarrow$ put($cnt_{act}, \kappa$);
        put($cnt_{fin}$, Nil);
        openBar $fin$;
        True
      False $\rightarrow$ put($cnt_{act}, act - 1$);
        let $\langle f, h \rangle$ = newhandled
        in put($cnt_{fin}, h : fin$); wait $h$
```

Figure 2.28: Encoding of a barrier.

The implementation of the barrier specification in $\lambda^{\tau}(\mathsf{fchb})$ is more concise about the internal functionality. It consists of three functions: newBar creates a new barrier. This 3-tuple consists of two buffers and the capacity value κ. One buffer is for the count of active threads, the other holds a list of already synchronised, waiting threads. The counting buffer initially is of value κ. With every syncBar-operation it will be decremented, because there is one active thread less. Furthermore syncBar creates a new handle component and adds it to the list of already synchronised threads. If the active count reaches 1, then the next syncBar operations is last one, thus it evaluates to the case where it uses openBar to bind all handles from the list to True before returning True to the calling thread. The types of the three functions are:

$$\begin{aligned}\mathsf{newBar} ::\ & \tau \to (\mathsf{buf}\ \tau, \mathsf{buf}\ (\mathsf{List\ bool}), \tau)\\ \mathsf{syncBar} ::\ & (\mathsf{buf}\ \tau, \mathsf{buf}\ (\mathsf{List\ bool}), \tau) \to \mathsf{bool}\\ \mathsf{openBar} ::\ & \mathsf{List}\ \tau \to \mathsf{unit}\end{aligned}$$

Translation From $\lambda^\tau(\mathsf{fchb}) + \mathsf{bar} \to \lambda^\tau(\mathsf{fchb})$.

$$T(x\ \mathsf{bar}\ \kappa) \mapsto (\nu a)(\nu f)(x \Leftarrow (a, f, \kappa)\ |\ a\ \mathsf{b}\ \kappa\ |\ f\ \mathsf{b}\ [\,])$$

Figure 2.29: Translation of components from $\lambda^\tau(\mathsf{fchb}) + \mathsf{bar}$ to $\lambda^\tau(\mathsf{fchb})$

In a translation T from $\lambda^\tau(\mathsf{fchb}) + \mathsf{bar}$ to $\lambda^\tau(\mathsf{fchb})$ a barrier component is mapped to the 3-tuple from the implementation (Fig. 2.29). Any barrier configuration is mapped to an empty barrier in $\lambda^\tau(\mathsf{fchb})$. The active thread counting buffer a is of value κ and the waiting queue of the second buffer f is empty. We define the following transformation rules:

(TBAR.NEW)

$$E[\mathsf{newBar}\ \kappa] \to (\nu x)(\nu a)(\nu f)(E[x]\ |\ x \Leftarrow (a, f, \kappa)\ |\ a\ \mathsf{b}\ \kappa\ |\ f\ \mathsf{b}\ [\,])$$

(TSYNC) for $0 < n \leq \kappa - 1$

$$\begin{aligned}&(\nu x)(\nu a)(\nu f)(\nu h_1, \ldots, h_n)(\nu f_1, \ldots, f_n)(E[\mathsf{syncBar}\ x]\\ &\ |\ x \Leftarrow (a, f, \kappa)\ |\ a\ \mathsf{b}\ \kappa - n\ |\ f\ \mathsf{b}\ [h_1 \ldots h_n]\ |\ h_1\ \mathsf{h}\ f_1\ |\ \ldots\ |\ h_n\ \mathsf{h}\ f_n)\\ &\to (\nu x)(\nu a)(\nu f)(\nu h_1, \ldots, h_{n+1})(\nu f_1, \ldots, f_{n+1})(E[h_{n+1}]\\ &\ |\ x \Leftarrow (a, f, \kappa)\ |\ a\ \mathsf{b}\ \kappa{-}n{-}1\ |\ f\ \mathsf{b}\ [h_1 \ldots h_{n+1}]\ |\ h_1\ \mathsf{h}\ f_1\ |\ \ldots\ |\ h_{n+1}\ \mathsf{h}\ f_{n+1})\end{aligned}$$

(TSYNC)

$$\begin{aligned}&(\nu x)(\nu a)(\nu f)(\nu h_1, \ldots, h_{\kappa-1})(\nu f_1, \ldots, f_{\kappa-1})(E[\mathsf{syncBar}\ x]\\ &\ |\ x \Leftarrow (a, f, \kappa)\ |\ a\ \mathsf{b}\ 1\ |\ f\ \mathsf{b}\ [h_1 \ldots h_{\kappa-1}]\ |\ h_1\ \mathsf{h}\ f_1\ |\ \ldots\ |\ h_{\kappa-1}\ \mathsf{h}\ f_{\kappa-1})\\ &\to (\nu x)(\nu a)(\nu f)(\nu h_1, \ldots, h_{\kappa-1})(\nu f_1, \ldots, f_{\kappa-1})(E[\mathsf{True}]\\ &\ |\ x \Leftarrow (a, f, \kappa)\ |\ a\ \mathsf{b}\ \kappa\ |\ f\ \mathsf{b}\ [\,]\ |\ h_1\ \mathsf{h}\ \bullet\ |\ \ldots\ |\ h_{\kappa-1}\ \mathsf{h}\ \bullet\ |\\ &f_1 \Leftarrow \mathsf{True}\ |\ \ldots\ |\ f_{\kappa-1} \Leftarrow \mathsf{True})\end{aligned}$$

(TBAR.NEW) creates new barrier components. (TSYNC) is a transformation rule for $\mathsf{syncBar}$, that evaluates to True if the active thread counting buffer is of value 1. Then the barrier returns to the state of newBar. In all the other cases $\mathsf{syncBar}$ swells in a new handle. The barrier type $\mathsf{bar}\ \tau$ maps to $(\mathsf{buf}\ T(\tau), \mathsf{buf}\ (\mathsf{List\ bool}), T(\tau))$.

Summary

In this section of chapter 2 we described more complex and more powerful concurrency primitives and denoted each of them as an extension for the $\lambda^{\tau}(\mathsf{fchb})$ calculus. We showed implementations of the primitive's operations in $\lambda^{\tau}(\mathsf{fchb})$ where we have used the resources of the language $\lambda^{\tau}(\mathsf{fchb})$. Each implementation led us to a translation of the primitive as a new abstraction in $\lambda^{\tau}(\mathsf{fchb})$. These new abstractions are now ready for examination of equivalence between each other in the next chapter.

3. Properties Of The Encodings

In this chapter we propose program equivalences based on the encodings from section 2.3. As mentioned in section 2.2.5 there are two ways for showing equivalence of two terms t, t' in $\lambda^{\tau}(\mathsf{fchb})$. One is to show equivalence between two processes and the other is based on equivalence between expressions. In both cases we must show that they are equal in terms of may-must convergence (Definition 2.2.4) for all contexts. Having done this we can say that t, t' are contextually equivalent.

Initially we show the properties of the operational behaviour of expressions in $\lambda^{\tau}(\mathsf{fchb})$. We investigate the reduction of expressions e_1 to e_2. If we only use correct reduction rules (Fig. 2.10) respectively correct transformation rules here, (Fig. 2.2.4) then the reduction from e_1 to e_2 is also correct and $e_1 \sim e_2$ holds. We use this method to show correctness of some transformation rules of new concurrency abstractions in $\lambda^{\tau}(\mathsf{fchb})$. Once we have proven correctness of a transformation rule, we may use it in further contexts as a correct rule.

In section 3.6 we show program equivalences in $\lambda^{\tau}(\mathsf{fchb})$. We investigate processes that use different concurrency abstractions that result in an equal behaviour. To prove this we must show that $p_1 \sim p_2$ holds. This proof has two parts: One is showing that $p_1 \to v$ and $p_2 \to v'$ are correct transformations. If $v = v'$ then it follows that $p_1 \sim p_2$. This is illustrated on the right.

$$\begin{tikzcd}
p_1 \arrow[r, dash, dashed, "\sim"] \arrow[d, "*"'] & p_2 \arrow[d, "*"'] \\
v \arrow[r, dash, dashed, "\sim"] & v'
\end{tikzcd}$$

To show that the translation of a calculus $\lambda^{\tau}(\mathsf{fchb}) + x$ is correct, we need to prove the translation $T : \lambda^{\tau}(\mathsf{fchb}) + x \to \lambda^{\tau}(\mathsf{fchb})$ as being adequate. This proof needs our properties of encodings in $\lambda^{\tau}(\mathsf{fchb})$ and should be a task for the future, not being discussed here. The illustration shows the coherence between programs of a new calculus and the $\lambda^{\tau}(\mathsf{fchb})$.

$$\begin{tikzcd}
p_1 \arrow[r, "*"] \arrow[d, mapsto, "T:\lambda^{\tau}(\mathsf{fchb})+x\to\lambda^{\tau}(\mathsf{fchb})"] & p_2 \arrow[d, mapsto] \\
T(p_1) \arrow[r, "*"] & T(p_2)
\end{tikzcd}$$

Please look at the table of abbreviations at page 13 that may help following the proofs.

In the proofs in section 3.6 we often use a new thread that reduces to $(\nu z)(E[z] \mid z \Leftarrow e)$ to show reduction of $E[e]$, where $e \in Exp$. This is allowed because of proposition 3.0.1.

Proposition 3.0.1.

$$\begin{aligned} E[e] \ &\sim\ (\nu z)(E[z] \mid z \Leftarrow e) \\ &\mathit{if}\ \ E[e] \xrightarrow{*} E[v] \end{aligned}$$

Proof. Given the precondition $E[e] \xrightarrow{*} E[v]$, then

$$\begin{aligned} & (\nu z)(E[z] \mid z \Leftarrow e) \\ \xrightarrow{*}\ & (\nu z)(E[z] \mid z \Leftarrow v) \\ \xrightarrow{\text{FUT.DEREF}}\ & (\nu z)(E[v] \mid z \Leftarrow v) \\ \xrightarrow{\text{GC}}\ & E[v] \end{aligned}$$

and

$$\begin{aligned} & E[v] \\ \xleftarrow{\text{GC}}\ & (\nu z)(E[v] \mid z \Leftarrow v) \\ \xleftarrow{\text{FUT.DEREF}}\ & (\nu z)(E[z] \mid z \Leftarrow v) \\ \xleftarrow{*}\ & (\nu z)(E[z] \mid z \Leftarrow e) \end{aligned}$$

Thus, $(\nu z)(E[z] \mid z \Leftarrow e) \xrightarrow{*} E[v]$.
Therefore $E[e] \ \sim\ (\nu z)(E[z] \mid z \Leftarrow e)$ holds. □

3.1 Properties Of The Buffer Operations

Our new concurrency primitives from section 3.6 are based on buffers. The correctness of the buffer implementation in $\lambda^\tau(\mathsf{fchb})$ had been shown in [SSNSS09]. So we can freely use reduction and transformation rules for buffers in our proofs.

Proposition 3.1.1.

$$E[\textbf{thread}\ (\lambda\_.\ \textit{let}\ b = (\textbf{newBuf}\ \textit{unit})\ \textit{in}\ \textbf{put}(b, v); \textbf{get}\ b)] \\ \sim\ E[v]$$

Proof. We begin the proof with the reduction of
$\textbf{thread}\ (\lambda\_.\ \mathsf{let}\ b = (\textbf{newBuf}\ \mathsf{unit})\ \mathsf{in}\ \textbf{put}(b, v); \textbf{get}\ b)$.
Applying THREAD.NEW:

$$(\nu z)(E[z]\ |\ z{\Leftarrow}(\lambda\_.\mathsf{let}\ b = (\textbf{newBuf}\ \mathsf{unit})\ \mathsf{in}\textbf{put}(b, v); \textbf{get}\ b)\ z)$$

Applying β-CBV(ev):

$$(\nu z)(E[z]\ |\ z{\Leftarrow}\mathsf{let}\ b = (\textbf{newBuf}\ \mathsf{unit})\ \mathsf{in}\ \textbf{put}(b, v); \textbf{get}\ b)$$

Removing let expression

$$(\nu z)(E[z]\ |\ z{\Leftarrow}(\lambda b.(\textbf{put}(b, v); \textbf{get}\ b))(\textbf{newBuf}\ \mathsf{unit}))$$

Removing sequential computation ;

$$(\nu z)(E[z]\ |\ z{\Leftarrow}(\lambda b.((\lambda\_.\textbf{get}\ b)(\textbf{put}(b, v))))(\textbf{newBuf}\ \mathsf{unit}))$$

Performing some reduction rules

$$\begin{aligned}
&\xrightarrow{\text{BUFF.NEW(ev)}} (\nu z)(\nu x_b)(E[z]\ |\ z{\Leftarrow}(\lambda b.(\lambda\_.(\textbf{get}\ b))(\textbf{put}(b, v)))x_b\ |\ x_b\,\mathsf{b}\,-)\\
&\xrightarrow{\beta\text{-CBV(ev)}} (\nu z)(\nu x_b)(E[z]\ |\ z{\Leftarrow}(\lambda\_.(\textbf{get}\ x_b))(\textbf{put}(x_b, v))\ |\ x_b\,\mathsf{b}\,-)\\
&\xrightarrow{\text{DET.PUT}} (\nu z)(\nu x_b)(E[z]\ |\ z{\Leftarrow}(\lambda\_.(\textbf{get}\ x_b))\ \mathsf{unit}\ |\ x_b\,\mathsf{b}\,v)\\
&\xrightarrow{\beta\text{-CBV(ev)}} (\nu z)(\nu x_b)(E[z]\ |\ z{\Leftarrow}\textbf{get}\ x_b\ |\ x_b\,\mathsf{b}\,v)\\
&\xrightarrow{\text{DET.GET}} (\nu z)(\nu x_b)(E[z]\ |\ z{\Leftarrow}v\ |\ x_b\,\mathsf{b}\,-)\\
&\xrightarrow{\text{GC}} E[v]
\end{aligned}$$

□

3.2 Properties Of The Channel Abstraction

We show correct transformations for the channel abstraction in $\lambda^{\tau}(\mathsf{fchb})$. We start with an expression where a channel operation is being used. Furthermore we reduce this expression until we reach an evaluation context for the specific channel operation. Here we replace the operation by its implementation. If both, the start of a proof and the result of the reduction, equal to a transformation rule's start and result, and if we only use correct reduction (resp. transformation) rules, then the transformation rule is also correct.

Proposition 3.2.1.

$$\begin{aligned}&E[\textsf{\textit{newChan unit}}]\\ \sim\ &(\nu x_h)(\nu x_r)(\nu x_w)(E[(x_r, x_w)] \mid x_w \,\mathsf{b}\, x_h \mid x_r \,\mathsf{b}\, x_h \mid x_h \,\mathsf{b}\, -)\end{aligned}$$

Proof. We begin giving the context $E[\mathsf{newChan\ unit}]$. Replacing the operation $\mathsf{newChan}$ by its implementation results in $\rightarrow$

$$\begin{aligned}E[(\lambda\_.\mathsf{let}\ hole &= (\textbf{newBuf}\ \mathsf{unit})\\ read &= (\textbf{newBuf}\ \mathsf{unit})\\ write &= (\textbf{newBuf}\ \mathsf{unit})\\ \mathsf{in}\ \ \textbf{put}(read, hole);&\textbf{put}(write, hole); (read, write))\mathsf{unit}]\end{aligned}$$

Removing some syntactical sugar $\mathsf{let} \ldots \mathsf{in}$ and the operator for sequential execution ; changes the term to:

$$\begin{gathered}E[\lambda\_.(\lambda hole.(\lambda read.(\lambda write.(\lambda\_.(\lambda\_.(read, write))\\ \textbf{put}(write, hole))\textbf{put}(read, hole))(\textbf{newBuf}\ \mathsf{unit}))(\textbf{newBuf}\ \mathsf{unit}))(\textbf{newBuf}\ \mathsf{unit}))\mathsf{unit}]\end{gathered}$$

Now we β-CBV(ev)-reduce unit on the outer λ-expression:

$$\begin{gathered}E[(\lambda hole.(\lambda read.(\lambda write.(\lambda\_.(\lambda\_.(read, write))\\ \textbf{put}(write, hole))\textbf{put}(read, hole))(\textbf{newBuf}\ \mathsf{unit}))(\textbf{newBuf}\ \mathsf{unit}))(\textbf{newBuf}\ \mathsf{unit})]\end{gathered}$$

We create a new buffer x_h using BUFF.NEW(ev):

$$\begin{gathered}(\nu x_h)(E[(\lambda hole.(\lambda read.(\lambda write.(\lambda\_.(\lambda\_.(read, write))\\ \textbf{put}(write, hole))\textbf{put}(read, hole))(\textbf{newBuf}\ \mathsf{unit}))(\textbf{newBuf}\ \mathsf{unit}))x_h] \mid x_h \,\mathsf{b}\, -)\end{gathered}$$

Binding x_h to $hole$ in the λ-expression via β-CBV(ev):

$$\begin{gathered}(\nu x_h)(E[(\lambda read.(\lambda write.(\lambda\_.(\lambda\_.(read, write))\\ \textbf{put}(write, x_h))\textbf{put}(read, x_h))(\textbf{newBuf}\ \mathsf{unit}))(\textbf{newBuf}\ \mathsf{unit})] \mid x_h \,\mathsf{b}\, -)\end{gathered}$$

Creating a new buffer x_r with BUFF.NEW(ev):

$$(\nu x_h)(\nu x_r)(E[(\lambda read.(\lambda write.(\lambda\_.(\lambda\_.(read, write)) \\ \mathbf{put}(write, x_h))\mathbf{put}(read, x_h))(\mathbf{newBuf}\ \mathsf{unit}))x_r] \mid x_r\,\mathsf{b}\,- \mid x_h\,\mathsf{b}\,-)$$

Binding x_r to *read* in the λ-expression via β-CBV(ev):

$$(\nu x_h)(\nu x_r)(E[(\lambda write.(\lambda\_.(\lambda\_.(x_r, write)) \\ \mathbf{put}(write, x_h))\mathbf{put}(x_r, x_h))(\mathbf{newBuf}\ \mathsf{unit})] \mid x_r\,\mathsf{b}\,- \mid x_h\,\mathsf{b}\,-)$$

$\xrightarrow{\text{BUFF.NEW(ev)}}$

$$(\nu x_h)(\nu x_r)(\nu x_w)(E[(\lambda write.(\lambda\_.(\lambda\_.(x_r, write)) \\ \mathbf{put}(write, x_h))\mathbf{put}(x_r, hole))x_w] \mid x_w\,\mathsf{b}\,- \mid x_r\,\mathsf{b}\,- \mid x_h\,\mathsf{b}\,-)$$

Binding x_w to *write* in the λ-expression via β-CBV(ev):

$$(\nu x_h)(\nu x_r)(\nu x_w)(E[(\lambda\_.(\lambda\_.(x_r, x_w)) \\ \mathbf{put}(x_w, x_h))\mathbf{put}(x_r, x_h)] \\ \mid x_w\,\mathsf{b}\,- \mid x_r\,\mathsf{b}\,- \mid x_h\,\mathsf{b}\,-)$$

Putting the hole x_h into buffer r with DET.PUT:

$$(\nu x_h)(\nu x_r)(\nu x_w)(E[(\lambda\_.(\lambda\_.(x_r, x_w))\mathbf{put}(x_w, x_h))\mathsf{unit}] \\ \mid x_w\,\mathsf{b}\,- \mid x_r\,\mathsf{b}\,x_h \mid x_h\,\mathsf{b}\,-)$$

$\xrightarrow{\beta\text{-CBV(ev)}}$

$$(\nu x_h)(\nu x_r)(\nu x_w)(E[(\lambda\_.(x_r, x_w))\mathbf{put}(x_w, x_h)] \\ \mid x_w\,\mathsf{b}\,- \mid x_r\,\mathsf{b}\,x_h \mid x_h\,\mathsf{b}\,-)$$

Putting the hole x_h into buffer w via DET.PUT:

$$(\nu x_h)(\nu x_r)(\nu x_w)(E[(\lambda\_.(x_r, x_w))\mathsf{unit}] \\ \mid x_w\,\mathsf{b}\,x_h \mid x_r\,\mathsf{b}\,x_h \mid x_h\,\mathsf{b}\,-)$$

Finally, beta-reducing the unit value with β-CBV(ev)leads to:

$$(\nu x_h)(\nu x_r)(\nu x_w)(E[(x_r, x_w)] \mid x_w\,\mathsf{b}\,x_h \mid x_r\,\mathsf{b}\,x_h \mid x_h\,\mathsf{b}\,-)$$

As we only used correct reduction and transformation rules BUFF.NEW(ev), BETA(ev), DET.PUTand GC, we conclude that this is also a correct transformation. □

Proposition 3.2.2.

$$(\nu x)(T(x\,\mathsf{ch}\,[v_1 \dots v_n]) \mid E[\mathit{writeChan}\ x\ v_{n+1}])$$
$$\sim\ (\nu x)(T(x\,\mathsf{ch}\,[v_1 \dots v_n, v_{n+1}]) \mid E[\mathit{unit}])$$

Proof. In this proposition we use the translation of the channel encoding for clarity. In our proof we translate the channel component into its representation in $\lambda^\tau(\mathsf{fchb})$. We start with $E(\mathsf{writeChan}\ x\ v)$, given a channel that already contains n items.

$$\begin{gathered}(\nu c)(\nu r)(\nu w)(\nu h_1 \dots h_{n+1})\\ (E[\mathsf{writeChan}\ c\ v_{n+1}] \mid c \Leftarrow (r,w) \mid r\,\mathsf{b}\,h_1 \mid h_1\,\mathsf{b}\,\mathsf{Item}(v_1,h_2)\\ \mid \dots \mid h_n\,\mathsf{b}\,\mathsf{Item}(v_n,h_{n+1}) \mid h_{n+1}\,\mathsf{b}\,- \mid w\,\mathsf{b}\,h_{n+1})\end{gathered}$$

Replacing the operation writeChan by its implementation:

$$\begin{gathered}(\nu c)(\nu r)(\nu w)(\nu h_1 \dots h_{n+1})\\ (E[(\lambda\langle read, write\rangle, v.\mathsf{let}\ newhole = (\mathbf{newBuf}\ \mathsf{unit}), oldhole = (\mathbf{get}\ write)\\ \mathsf{in}\ \mathbf{put}(write, newhole); \mathbf{put}(oldhole, \mathsf{Item}(v, newhole)))\ c\ v_{n+1}]\\ \mid c \Leftarrow (r,w) \mid r\,\mathsf{b}\,h_1 \mid h_1\,\mathsf{b}\,\mathsf{Item}(v_1,h_2)\\ \mid \dots \mid h_n\,\mathsf{b}\,\mathsf{Item}(v_n,h_{n+1}) \mid h_{n+1}\,\mathsf{b}\,- \mid w\,\mathsf{b}\,h_{n+1})\end{gathered}$$

We remove the guard $\lambda\langle read, write\rangle$:

$$\begin{gathered}(\nu c)(\nu r)(\nu w)(\nu h_1 \dots h_{n+1})\\ (E[(\lambda x.\ \mathbf{case}\ x\ \mathbf{of}\langle read, write\rangle \to\\ \lambda v.\mathsf{let}\ newhole = (\mathbf{newBuf}\ \mathsf{unit}), oldhole = (\mathbf{get}\ write)\\ \mathsf{in}\ \mathbf{put}(write, newhole); \mathbf{put}(oldhole, \mathsf{Item}(v, newhole)))\ c\ v_{n+1}]\\ \mid c \Leftarrow (r,w) \mid r\,\mathsf{b}\,h_1 \mid h_1\,\mathsf{b}\,\mathsf{Item}(v_1,h_2)\\ \mid \dots \mid h_n\,\mathsf{b}\,\mathsf{Item}(v_n,h_{n+1}) \mid h_{n+1}\,\mathsf{b}\,- \mid w\,\mathsf{b}\,h_{n+1})\end{gathered}$$

We β-CBV(ev)-reduce c to the λ-expression using:

$$\begin{gathered}(\nu c)(\nu r)(\nu w)(\nu h_1 \dots h_{n+1})\\ (E[(\mathbf{case}\ c\ \mathbf{of}\ \langle read, write\rangle \to\\ \lambda v.\mathsf{let}\ newhole = (\mathbf{newBuf}\ \mathsf{unit}), oldhole = (\mathbf{get}\ write)\\ \mathsf{in}\ \ \mathbf{put}(write, newhole); \mathbf{put}(oldhole, \mathsf{Item}(v, newhole)))\ v_{n+1}]\\ \mid c \Leftarrow (r,w) \mid r\,\mathsf{b}\,h_1 \mid h_1\,\mathsf{b}\,\mathsf{Item}(v_1,h_2)\\ \mid \dots \mid h_n\,\mathsf{b}\,\mathsf{Item}(v_n,h_{n+1}) \mid h_{n+1}\,\mathsf{b}\,- \mid w\,\mathsf{b}\,h_{n+1})\end{gathered}$$

We replace c in the case-expression with a dereference of its value by FUT.DEREF(ev):

$$\begin{gathered}(\nu c)(\nu r)(\nu w)(\nu h_1 \ldots h_{n+1})\\ (E[(\textbf{case}\ (r,w)\ \textbf{of}\ \langle read, write\rangle \to\\ \lambda v.\mathsf{let}\ newhole = (\textbf{newBuf}\ \mathsf{unit}), oldhole = (\textbf{get}\ write)\\ \mathsf{in}\ \textbf{put}(write, newhole); \textbf{put}(oldhole, \mathsf{Item}(v, newhole)))\ v_{n+1}]\\ \mid c \Leftarrow (r,w) \mid r\,\mathsf{b}\,h_1 \mid h_1\,\mathsf{b}\,\mathsf{Item}(v_1,h_2)\\ \mid \ \ldots \mid h_n\,\mathsf{b}\,\mathsf{Item}(v_n,h_{n+1}) \mid h_{n+1}\,\mathsf{b}\ - \ \mid w\,\mathsf{b}\,h_{n+1})\end{gathered}$$

Now we can apply CASE.BETA(ev):

$$\begin{gathered}(\nu c)(\nu r)(\nu w)(\nu h_1 \ldots h_{n+1})\\ (E[(\lambda v.\mathsf{let}\ newhole = (\textbf{newBuf}\ \mathsf{unit}), oldhole = (\textbf{get}\ w)\\ \mathsf{in}\ \textbf{put}(w, newhole); \textbf{put}(oldhole, \mathsf{Item}(v, newhole)))\ v_{n+1}]\\ \mid c \Leftarrow (r,w) \mid r\,\mathsf{b}\,h_1 \mid h_1\,\mathsf{b}\,\mathsf{Item}(v_1,h_2)\\ \mid \ \ldots \mid h_n\,\mathsf{b}\,\mathsf{Item}(v_n,h_{n+1}) \mid h_{n+1}\,\mathsf{b}\ - \ \mid w\,\mathsf{b}\,h_{n+1})\end{gathered}$$

We replace v by v_{n+1} in the λ-expression with β-CBV(ev):

$$\begin{gathered}(\nu c)(\nu r)(\nu w)(\nu h_1 \ldots h_{n+1})\\ (E[\mathsf{let}\ newhole = (\textbf{newBuf}\ \mathsf{unit}), oldhole = (\textbf{get}\ w)\\ \mathsf{in}\ \textbf{put}(w, newhole); \textbf{put}(oldhole, \mathsf{Item}(v_{n+1}, newhole))]\\ \mid c \Leftarrow (r,w) \mid r\,\mathsf{b}\,h_1 \mid h_1\,\mathsf{b}\,\mathsf{Item}(v_1,h_2)\\ \mid \ \ldots \mid h_n\,\mathsf{b}\,\mathsf{Item}(v_n,h_{n+1}) \mid h_{n+1}\,\mathsf{b}\ - \ \mid w\,\mathsf{b}\,h_{n+1})\end{gathered}$$

Replacing let . . . in :

$$\begin{gathered}(\nu c)(\nu r)(\nu w)(\nu h_1 \ldots h_{n+1})\\ (E[(\lambda\ newhole.(\lambda\ oldhole.\\ \textbf{put}(w, newhole); \textbf{put}(oldhole, \mathsf{Item}(v_{n+1}, newhole)))(\textbf{get}\ w))(\textbf{newBuf}\ \mathsf{unit})]\\ \mid c \Leftarrow (r,w) \mid r\,\mathsf{b}\,h_1 \mid h_1\,\mathsf{b}\,\mathsf{Item}(v_1,h_2)\\ \mid \ \ldots \mid h_n\,\mathsf{b}\,\mathsf{Item}(v_n,h_{n+1}) \mid h_{n+1}\,\mathsf{b}\ - \ \mid w\,\mathsf{b}\,h_{n+1})\end{gathered}$$

Creating a new buffer h_{n+2} with BUFF.NEW(ev) $\to$

$$\begin{gathered}(\nu c)(\nu r)(\nu w)(\nu h_1 \ldots h_{n+1}, h_{n+2})\\ (E[(\lambda\ newhole.(\lambda\ oldhole.\\ \textbf{put}(w, newhole); \textbf{put}(oldhole, \mathsf{Item}(v_{n+1}, newhole)))(\textbf{get}\ w))h_{n+2}]\\ \mid c \Leftarrow (r,w) \mid r\,\mathsf{b}\,h_1 \mid h_1\,\mathsf{b}\,\mathsf{Item}(v_1,h_2)\\ \mid \ \ldots \mid h_n\,\mathsf{b}\,\mathsf{Item}(v_n,h_{n+1}) \mid h_{n+1}\,\mathsf{b}\ - \ \mid h_{n+2}\,\mathsf{b}\ - \ \mid w\,\mathsf{b}\,h_{n+1})\end{gathered}$$

β-CBV(ev) replaces *newhole* by h_{n+2}:

$$\begin{gathered}(\nu c)(\nu r)(\nu w)(\nu h_1 \ldots h_{n+1}, h_{n+2})\\(E[(\lambda\ oldhole.\mathbf{put}(w, h_{n+2}); \mathbf{put}(oldhole, \mathsf{Item}(v_{n+1}, h_{n+2})))(\mathbf{get}\ w)]\\ \mid c \Leftarrow (r, w) \mid r \,\mathsf{b}\, h_1 \mid h_1 \,\mathsf{b}\, \mathsf{Item}(v_1, h_2)\\ \mid \ldots \mid h_n \,\mathsf{b}\, \mathsf{Item}(v_n, h_{n+1}) \mid h_{n+1} \,\mathsf{b}\, - \mid h_{n+2} \,\mathsf{b}\, - \mid w \,\mathsf{b}\, h_{n+1})\end{gathered}$$

Now we can get the contents of buffer w with DET.GET:

$$\begin{gathered}(\nu c)(\nu r)(\nu w)(\nu h_1 \ldots h_{n+1}, h_{n+2})\\(E[(\lambda\ oldhole.\mathbf{put}(w, h_{n+2}); \mathbf{put}(oldhole, \mathsf{Item}(v_{n+1}, h_{n+2})))h_{n+1}]\\ \mid c \Leftarrow (r, w) \mid r \,\mathsf{b}\, h_1 \mid h_1 \,\mathsf{b}\, \mathsf{Item}(v_1, h_2)\\ \mid \ldots \mid h_n \,\mathsf{b}\, \mathsf{Item}(v_n, h_{n+1}) \mid h_{n+1} \,\mathsf{b}\, - \mid h_{n+2} \,\mathsf{b}\, - \mid w \,\mathsf{b}\, -)\end{gathered}$$

β-CBV(ev) replaces *oldhole* by h_{n+1}:

$$\begin{gathered}(\nu c)(\nu r)(\nu w)(\nu h_1 \ldots h_{n+1}, h_{n+2})\\(E[\mathbf{put}(w, h_{n+2}); \mathbf{put}(h_{n+1}, \mathsf{Item}(v_{n+1}, h_{n+2}))]\\ \mid c \Leftarrow (r, w) \mid r \,\mathsf{b}\, h_1 \mid h_1 \,\mathsf{b}\, \mathsf{Item}(v_1, h_2)\\ \mid \ldots \mid h_n \,\mathsf{b}\, \mathsf{Item}(v_n, h_{n+1}) \mid h_{n+1} \,\mathsf{b}\, - \mid h_{n+2} \,\mathsf{b}\, - \mid w \,\mathsf{b}\, -)\end{gathered}$$

DET.PUT writes h_{n+2} into buffer w:

$$\begin{gathered}(\nu c)(\nu r)(\nu w)(\nu h_1 \ldots h_{n+1}, h_{n+2})\\(E[\mathsf{unit}; \mathbf{put}(h_{n+1}, \mathsf{Item}(v_{n+1}, h_{n+2}))]\\ \mid c \Leftarrow (r, w) \mid r \,\mathsf{b}\, h_1 \mid h_1 \,\mathsf{b}\, \mathsf{Item}(v_1, h_2)\\ \mid \ldots \mid h_n \,\mathsf{b}\, \mathsf{Item}(v_n, h_{n+1}) \mid h_{n+1} \,\mathsf{b}\, - \mid h_{n+2} \,\mathsf{b}\, - \mid w \,\mathsf{b}\, h_{n+2})\end{gathered}$$

Removing sequential notation:

$$\begin{gathered}(\nu c)(\nu r)(\nu w)(\nu h_1 \ldots h_{n+1}, h_{n+2})\\(E[(\lambda\_.\mathbf{put}(h_{n+1}, \mathsf{Item}(v_{n+1}, h_{n+2})))\mathsf{unit}]\\ \mid c \Leftarrow (r, w) \mid r \,\mathsf{b}\, h_1 \mid h_1 \,\mathsf{b}\, \mathsf{Item}(v_1, h_2)\\ \mid \ldots \mid h_n \,\mathsf{b}\, \mathsf{Item}(v_n, h_{n+1}) \mid h_{n+1} \,\mathsf{b}\, - \mid h_{n+2} \,\mathsf{b}\, - \mid w \,\mathsf{b}\, h_{n+2})\end{gathered}$$

β-CBV(ev) of unit:

$$\begin{gathered}(\nu c)(\nu r)(\nu w)(\nu h_1 \ldots h_{n+1}, h_{n+2})\\(E[\mathbf{put}(h_{n+1}, \mathsf{Item}(v_{n+1}, h_{n+2}))]\\ \mid c \Leftarrow (r, w) \mid r \,\mathsf{b}\, h_1 \mid h_1 \,\mathsf{b}\, \mathsf{Item}(v_1, h_2)\\ \mid \ldots \mid h_n \,\mathsf{b}\, \mathsf{Item}(v_n, h_{n+1}) \mid h_{n+1} \,\mathsf{b}\, - \mid h_{n+2} \,\mathsf{b}\, - \mid w \,\mathsf{b}\, h_{n+2})\end{gathered}$$

DET.PUT writes a new item (v_{n+1}, h_{n+2}) into the empty buffer h_{n+1}:

$$\begin{gathered}(\nu c)(\nu r)(\nu w)(\nu h_1 \ldots h_{n+1}, h_{n+2})\\ (E[\mathsf{unit}] \mid c \Leftarrow (r,w) \mid r \,\mathsf{b}\, h_1 \mid h_1 \,\mathsf{b}\, \mathsf{Item}(v_1, h_2)\\ \mid \ldots \mid h_n \,\mathsf{b}\, \mathsf{Item}(v_n, h_{n+1}) \mid h_{n+1} \,\mathsf{b}\, \mathsf{Item}(v_{n+1}, h_{n+2}) \mid h_{n+2} \,\mathsf{b}\, - \mid w \,\mathsf{b}\, h_{n+2})\end{gathered}$$

Finally we apply the translation T on the channel:

$$(\nu x)(E[\mathsf{unit}] \mid T(x \,\mathsf{ch}\, [v_1 \ldots v_n, v_{n+1}]))$$

This is also the result of the transformation rule TCHAN.WRITE. As we only used correct reduction and transformation rules BUFF.NEW(ev), BETA(ev), DET.PUTand GC, we conclude that TCHAN.WRITE is also a correct transformation rule. □

Proposition 3.2.3.

$$\begin{gathered}(\nu x)(T(x \,\mathsf{ch}\, [v_1 \ldots v_n]) \mid E[\mathit{writeChan}\ x\ (w_1 \ldots w_m)])\\ \sim\ (\nu x)(T(x \,\mathsf{ch}\, [v_1 \ldots v_n, w_1 \ldots w_m]) \mid E[\mathit{unit}])\end{gathered}$$

for all finite n, m.

Proof. Our induction base is 3.2.2. To prove the induction hypothesis 3.2.3 we add an induction step: m → m+1:

$$\begin{gathered}(\nu c)(\nu r)(\nu w)(\nu h_1 \ldots h_{n+1})\\ (E[\mathsf{writeChan}\ x\ (w_1 \ldots w_m, w_{m+1})] \mid c \Leftarrow (r,w) \mid r \,\mathsf{b}\, h_1 \mid h_1 \,\mathsf{b}\, \mathsf{Item}(v_1, h_2)\\ \mid \ldots \mid h_n \,\mathsf{b}\, \mathsf{Item}(v_n, h_{n+1}) \mid h_{n+1} \,\mathsf{b}\, - \mid w \,\mathsf{b}\, h_{n+1})\end{gathered}$$

As stated in section 2.3, the channel encoding, we use writeChan $c\ (v_1 \ldots v_n)$ as new shorthand for writeChan $c\ v_1$; writeChan $c\ v_2; \ldots$; writeChan $c\ v_n$ here. Given that, we can split the writeChan operation:

$$\begin{gathered}(\nu c)(\nu r)(\nu w)(\nu h_1 \ldots h_{n+1})\\ (E[\mathsf{writeChan}\ c\ w_1; \mathsf{writeChan}\ c\ (w_2, \ldots, w_{m+1})]\\ \mid c \Leftarrow (r,w) \mid r \,\mathsf{b}\, h_1 \mid h_1 \,\mathsf{b}\, \mathsf{Item}(v_1, h_2)\\ \mid \ldots \mid h_n \,\mathsf{b}\, \mathsf{Item}(v_n, h_{n+1}) \mid h_{n+1} \,\mathsf{b}\, - \mid w \,\mathsf{b}\, h_{n+1})\end{gathered}$$

Performing writeChan $c\ w_1$ using the induction base:

$$\begin{gathered}(\nu c)(\nu r)(\nu w)(\nu h_1 \ldots h_{n+2})\\ (E[\mathsf{unit}; \mathsf{writeChan}\ c\ (w_2, \ldots, w_{m+1})]\\ \mid c \Leftarrow (r,w) \mid r \,\mathsf{b}\, h_1 \mid h_1 \,\mathsf{b}\, \mathsf{Item}(v_1, h_2)\\ \mid \ldots \mid h_n \,\mathsf{b}\, \mathsf{Item}(v_n, h_{n+1}) \mid h_{n+1} \,\mathsf{b}\, \mathsf{Item}(w_1, h_{n+2}) \mid h_{n+2} \,\mathsf{b}\, - \mid w \,\mathsf{b}\, h_{n+2})\end{gathered}$$

Performing $\mathsf{writeChan}\ c\ (w_2, \ldots, w_{m+1})$ using the induction hypothesis results in:

$$\begin{gathered}(\nu c)(\nu r)(\nu r)(\nu w)(\nu h_1 \ldots h_n, h_{n+m+2})(E[\mathsf{unit}; \mathsf{unit}] \\ \mid c \Leftarrow (r, w) \mid r\, \mathsf{b}\, h_1 \mid h_1\, \mathsf{b}\, \mathsf{Item}(v_1, h_2) \\ \mid \ldots \mid h_n\, \mathsf{b}\, \mathsf{Item}(v_n, h_{n+1}) \mid h_{n+1}\, \mathsf{b}\, \mathsf{Item}(w_1, h_{n+2}) \mid \ldots \mid \\ h_{n+m+1}\, \mathsf{b}\, \mathsf{Item}(w_{m+1}, h_{n+m+2}) \mid h_{n+m+2}\, \mathsf{b}\, - \mid w\, \mathsf{b}\, h_{n+m+2})\end{gathered}$$

Removing sequential computation changes the term to $(\ldots E[(\lambda\_.\mathsf{unit})\mathsf{unit}] \mid \ldots)$. Performing β-CBV(ev) leads to:

$$\begin{gathered}(\nu c)(\nu r)(\nu r)(\nu w)(\nu h_1 \ldots h_n, h_{n+m+1})(E[\mathsf{unit}] \mid \\ \mid c \Leftarrow (r, w) \mid r\, \mathsf{b}\, h_1 \mid h_1\, \mathsf{b}\, \mathsf{Item}(v_1, h_2) \\ \mid \ldots \mid h_n\, \mathsf{b}\, \mathsf{Item}(v_n, h_{n+1}) \mid h_{n+1}\, \mathsf{b}\, \mathsf{Item}(w_1, h_{n+2}) \mid \ldots \mid \\ h_{n+m+1}\, \mathsf{b}\, \mathsf{Item}(w_{m+1}, h_{n+m+2}) \mid h_{n+m+2}\, \mathsf{b}\, - \mid w\, \mathsf{b}\, h_{n+m+2})\end{gathered}$$

And this is exactly:

$$(\nu x)(T(x\, \mathsf{ch}\, [v_1 \ldots v_n, w_1 \ldots w_{m+1}]) \mid E[\mathsf{unit}])$$

With that we have proven the assumption. TCHAN.WRITEN is a correct program transformation in $\lambda^\tau(\mathsf{fchb})$. □

Proposition 3.2.4.

$$\begin{gathered}(\nu z)(\nu c)(\nu r)(\nu w)(\nu h_1, \ldots, h_n) \\ (E[\mathit{readChan}\ c] \mid c \Leftarrow (r, w) \mid r\, \mathsf{b}\, h_1 \mid h_1\, \mathsf{b}\, \mathit{Item}(v_1, h_2) \mid \ldots \\ \mid h_{n-1}\, \mathsf{b}\, \mathit{Item}(v_{n-1}, h_n) \mid h_n\, \mathsf{b}\, - \mid w\, \mathsf{b}\, h_n) \\ \sim \\ (\nu z)(\nu c)(\nu r)(\nu w)(\nu h_2, \ldots, h_n) \\ (E[v_1] \mid c \Leftarrow (r, w) \mid r\, \mathsf{b}\, h_2 \mid h_2\, \mathsf{b}\, \mathit{Item}(v_2, h_3) \mid \ldots \\ \mid h_{n-1}\, \mathsf{b}\, \mathit{Item}(v_{n-1}, h_n) \mid h_n\, \mathsf{b}\, - \mid w\, \mathsf{b}\, h_n)\end{gathered}$$

for $n \geq 0$

Proof. We start with the following program:

$$\begin{gathered}(\nu c)(\nu r)(\nu w)(\nu h_1, \ldots, h_n) \\ (E[\mathsf{readChan}\ c] \mid c \Leftarrow (r, w) \mid r\, \mathsf{b}\, h_1 \mid h_1\, \mathsf{b}\, \mathsf{Item}(v_1, h_2) \mid \ldots \\ \mid h_{n-1}\, \mathsf{b}\, \mathsf{Item}(v_{n-1}, h_n) \mid h_n\, \mathsf{b}\, - \mid w\, \mathsf{b}\, h_n)\end{gathered}$$

Removing the operation $\mathsf{readChan}$ by its implementation results in:

$$\begin{gathered}
(\nu c)(\nu r)(\nu w)(\nu h_1,\dots,h_n)\\
(E[(\lambda\langle read, write\rangle\_.\\
\mathsf{let}\ r' = (\textbf{get}\ read)\\
hd = (\textbf{get}\ r')\\
\mathsf{in}\ \ \textbf{case}\ hd\ \textbf{of}\ \mathsf{Item}(res, tl) \to \textbf{put}(read, tl); res)\ c]\\
\mid c \Leftarrow (r, w) \mid r\,\mathsf{b}\,h_1 \mid h_1\,\mathsf{b}\,\mathsf{Item}(v_1, h_2) \mid \dots\\
\mid h_{n-1}\,\mathsf{b}\,\mathsf{Item}(v_{n-1}, h_n) \mid h_n\,\mathsf{b}\ - \ \mid w\,\mathsf{b}\,h_n)
\end{gathered}$$

Removing the guard $\lambda\langle read, write\rangle\_.$ and the let-expression:

$$\begin{gathered}
(\nu c)(\nu r)(\nu w)(\nu h_1,\dots,h_n)\\
(E[(\lambda y.\textbf{case}\ y\ \textbf{of}\ \langle read, write\rangle \to\\
(\lambda r'(\lambda\ hd(\textbf{case}\ hd\ \textbf{of}\ \mathsf{Item}(res, tl) \to \textbf{put}(read, tl); res))(\textbf{get}\ r'))(\textbf{get}\ read))c]\\
\mid c \Leftarrow (r, w) \mid r\,\mathsf{b}\,h_1 \mid h_1\,\mathsf{b}\,\mathsf{Item}(v_1, h_2) \mid \dots\\
\mid h_{n-1}\,\mathsf{b}\,\mathsf{Item}(v_{n-1}, h_n) \mid h_n\,\mathsf{b}\ - \ \mid w\,\mathsf{b}\,h_n)
\end{gathered}$$

β-CBV(ev) of c:

$$\begin{gathered}
(\nu c)(\nu r)(\nu w)(\nu h_1,\dots,h_n)\\
(E[\textbf{case}\ c\ \textbf{of}\ \langle read, write\rangle \to\\
(\lambda r'\ (\lambda\ hd(\textbf{case}\ hd\ \textbf{of}\ \mathsf{Item}(res, tl) \to \textbf{put}(read, tl); res))(\textbf{get}\ r'))(\textbf{get}\ read)]\\
\mid c \Leftarrow (r, w) \mid r\,\mathsf{b}\,h_1 \mid h_1\,\mathsf{b}\,\mathsf{Item}(v_1, h_2) \mid \dots\\
\mid h_{n-1}\,\mathsf{b}\,\mathsf{Item}(v_{n-1}, h_n) \mid h_n\,\mathsf{b}\ - \ \mid w\,\mathsf{b}\,h_n)
\end{gathered}$$

We get the contents of c with FUT.DEREF(ev):

$$\begin{gathered}
(\nu c)(\nu r)(\nu w)(\nu h_1,\dots,h_n)\\
(E[\textbf{case}\ (r, w)\ \textbf{of}\ \langle read, write\rangle \to\\
(\lambda r'\ (\lambda\ hd(\textbf{case}\ hd\ \textbf{of}\ \mathsf{Item}(res, tl) \to \textbf{put}(read, tl); res))(\textbf{get}\ r'))(\textbf{get}\ read)]\\
\mid c \Leftarrow (r, w) \mid r\,\mathsf{b}\,h_1 \mid h_1\,\mathsf{b}\,\mathsf{Item}(v_1, h_2) \mid \dots\\
\mid h_{n-1}\,\mathsf{b}\,\mathsf{Item}(v_{n-1}, h_n) \mid h_n\,\mathsf{b}\ - \ \mid w\,\mathsf{b}\,h_n)
\end{gathered}$$

Now the case-expression is evaluable through CASE.BETA(ev):

$$\begin{gathered}
(\nu c)(\nu r)(\nu w)(\nu h_1,\dots,h_n)\\
(E[(\lambda r'(\lambda\ hd(\textbf{case}\ hd\ \textbf{of}\ \mathsf{Item}(res, tl) \to\\
\textbf{put}(r, tl); res))(\textbf{get}\ r'))(\textbf{get}\ r)]\\
\mid c \Leftarrow (r, w) \mid r\,\mathsf{b}\,h_1 \mid h_1\,\mathsf{b}\,\mathsf{Item}(v_1, h_2) \mid \dots\\
\mid h_{n-1}\,\mathsf{b}\,\mathsf{Item}(v_{n-1}, h_n) \mid h_n\,\mathsf{b}\ - \ \mid w\,\mathsf{b}\,h_n)
\end{gathered}$$

We empty the buffer r with DET.GET and get its content h_1:

$$\begin{gathered}(\nu c)(\nu r)(\nu w)(\nu h_1,\ldots,h_n)\\ (E[(\lambda r'(\lambda\ hd(\textbf{case}\ hd\ \textbf{of}\ \mathsf{Item}(res,tl)\rightarrow\\ \textbf{put}(r,tl);res))(\textbf{get}\ r'))h_1]\\ \mid c\Leftarrow(r,w)\mid r\,\mathsf{b}\,-\ \mid h_1\,\mathsf{b}\,\mathsf{Item}(v_1,h_2)\mid\ldots\\ \mid h_{n-1}\,\mathsf{b}\,\mathsf{Item}(v_{n-1},h_n)\mid h_n\,\mathsf{b}\,-\ \mid w\,\mathsf{b}\,h_n)\end{gathered}$$

$\xrightarrow{\beta\text{-CBV}(\mathsf{ev})}$

$$\begin{gathered}(\nu c)(\nu r)(\nu w)(\nu h_1,\ldots,h_n)\\ (E[(\lambda hd(\textbf{case}\ hd\ \textbf{of}\ \mathsf{Item}(res,tl)\rightarrow\\ \textbf{put}(r,tl);res))(\textbf{get}\ h_1)]\\ \mid c\Leftarrow(r,w)\mid r\,\mathsf{b}\,-\ \mid h_1\,\mathsf{b}\,\mathsf{Item}(v_1,h_2)\mid\ldots\\ \mid h_{n-1}\,\mathsf{b}\,\mathsf{Item}(v_{n-1},h_n)\mid h_n\,\mathsf{b}\,-\ \mid w\,\mathsf{b}\,h_n)\end{gathered}$$

The next step empties the buffer h_1 and gets the contents (DET.GET):

$$\begin{gathered}(\nu c)(\nu r)(\nu w)(\nu h_1,\ldots,h_n)\\ (E[(\lambda hd(\textbf{case}\ hd\ \textbf{of}\ \mathsf{Item}(res,tl)\rightarrow\\ \textbf{put}(r,tl);res))\mathsf{Item}(v_1,h_2)]\\ \mid c\Leftarrow(r,w)\mid r\,\mathsf{b}\,-\ \mid h_1\,\mathsf{b}\,-\ \mid h_2\,\mathsf{b}\,\mathsf{Item}(v_2,h_3)\mid\ldots\\ \mid h_{n-1}\,\mathsf{b}\,\mathsf{Item}(v_{n-1},h_n)\mid h_n\,\mathsf{b}\,-\ \mid w\,\mathsf{b}\,h_n)\end{gathered}$$

β-CBV(ev) replaces hd by $\mathsf{Item}(v_1,h_2)$:

$$\begin{gathered}(\nu c)(\nu r)(\nu w)(\nu h_1,\ldots,h_n)\\ (E[(\textbf{case}\ \mathsf{Item}(v_1,h_2)\ \textbf{of}\ \mathsf{Item}(res,tl)\rightarrow\textbf{put}(r,tl);res)]\\ \mid c\Leftarrow(r,w)\mid r\,\mathsf{b}\,-\ \mid h_1\,\mathsf{b}\,-\ \mid h_2\,\mathsf{b}\,\mathsf{Item}(v_2,h_3)\mid\ldots\\ \mid h_{n-1}\,\mathsf{b}\,\mathsf{Item}(v_{n-1},h_n)\mid h_n\,\mathsf{b}\,-\ \mid w\,\mathsf{b}\,h_n)\end{gathered}$$

$\xrightarrow{\text{CASE.BETA}(\mathsf{ev})}$

$$\begin{gathered}(\nu c)(\nu r)(\nu w)(\nu h_1,\ldots,h_n)\\ (E[\textbf{put}(r,h_2);v_1]\\ \mid c\Leftarrow(r,w)\mid r\,\mathsf{b}\,-\ \mid h_1\,\mathsf{b}\,-\ \mid h_2\,\mathsf{b}\,\mathsf{Item}(v_2,h_3)\mid\ldots\\ \mid h_{n-1}\,\mathsf{b}\,\mathsf{Item}(v_{n-1},h_n)\mid h_n\,\mathsf{b}\,-\ \mid w\,\mathsf{b}\,h_n)\end{gathered}$$

Putting h_2 into the buffer r with DET.PUT:

$$\begin{gathered}(\nu c)(\nu r)(\nu w)(\nu h_1,\ldots,h_n)\\ (E[\mathsf{unit};v_1]\\ \mid c\Leftarrow(r,w)\mid r\,\mathsf{b}\,h_2\mid h_1\,\mathsf{b}\,-\ \mid h_2\,\mathsf{b}\,\mathsf{Item}(v_2,h_3)\mid\ldots\\ \mid h_{n-1}\,\mathsf{b}\,\mathsf{Item}(v_{n-1},h_n)\mid h_n\,\mathsf{b}\,-\ \mid w\,\mathsf{b}\,h_n)\end{gathered}$$

We do some sequential computation removement $\xrightarrow{*}$

$$\begin{gathered}(\nu c)(\nu r)(\nu w)(\nu h_1, \dots, h_n)\\ (E[v_1]\\ \mid c \Leftarrow (r, w) \mid r \,\mathsf{b}\, h_2 \mid h_1 \,\mathsf{b}\, - \mid h_2 \,\mathsf{b}\, \mathsf{Item}(v_2, h_3) \mid \dots\\ \mid h_{n-1} \,\mathsf{b}\, \mathsf{Item}(v_{n-1}, h_n) \mid h_n \,\mathsf{b}\, - \mid w \,\mathsf{b}\, h_n)\end{gathered}$$

In a final step the garbage collection removes the empty buffer h_1. $\xrightarrow{\text{GC}}$

$$\begin{gathered}(\nu c)(\nu r)(\nu w)(\nu h_2, \dots, h_n)\\ (E[v_1] \mid c \Leftarrow (r, w) \mid r \,\mathsf{b}\, h_2 \mid h_2 \,\mathsf{b}\, \mathsf{Item}(v_2, h_3) \mid \dots\\ \mid h_{n-1} \,\mathsf{b}\, \mathsf{Item}(v_{n-1}, h_n) \mid h_n \,\mathsf{b}\, - \mid w \,\mathsf{b}\, h_n)\end{gathered}$$

This is also the result for the transformation rule TCHAN.READ. As we only used correct reduction or transformation rules we conclude that TCHAN.READ is also a correct transformation rule. □

Theorem 3 (Correctness of transformation rules of the channel abstraction in $\lambda^\tau(\mathsf{fchb})$)**.** TCHAN.NEW, TCHAN.WRITE, TCHAN.WRITEN *and* TCHAN.READ *are correct transformation rules in* $\lambda^\tau(\mathsf{fchb})$.

Proof. The correctness of TCHAN.NEW follows from proposition 3.2.1, as the result $(\nu x_h)(\nu x_r)(\nu x_w)(E[(x_r, x_w)] \mid x_w \,\mathsf{b}\, x_h \mid x_r \,\mathsf{b}\, x_h \mid x_h \,\mathsf{b}\, -)$ can reversely be transformed:
$\xleftarrow{\text{GC}}$

$$(\nu c)(\nu x_h)(\nu x_r)(\nu x_w)(E[(x_r, x_w)] \mid c{\Leftarrow}(x_r, x_w) \mid x_w \,\mathsf{b}\, x_h \mid x_r \,\mathsf{b}\, x_h \mid x_h \,\mathsf{b}\, -)$$

$\xleftarrow{\text{FUT.DEREF}}$

$$(\nu c)(\nu x_h)(\nu x_r)(\nu x_w)(E[c] \mid c{\Leftarrow}(x_r, x_w) \mid x_w \,\mathsf{b}\, x_h \mid x_r \,\mathsf{b}\, x_h \mid x_h \,\mathsf{b}\, -)$$

And this is exactly the resulting term of the rule TCHAN.NEW.

The correctness of TCHAN.WRITE, TCHAN.WRITEN and TCHAN.READ follows from propositions 3.2.2, 3.2.3 and 3.2.4. □

3.3 Properties Of The Quantity Semaphore Abstraction

We show correct transformation rules for the quantity semaphore abstraction in $\lambda^\tau(\mathsf{fchb})$. We proceed as aforementioned: We start with an expression using an operation on a quantity semaphore. We reduce it and compare the result with the resulting expression from the related transformation rule. If this holds and we only used correct reduction (resp. transformation) rules, then the transformation rule is also correct. We examine the operations up and down according to the implementation with handles.

Proposition 3.3.1.

$$E[\mathit{newQSem}\ \kappa] \ \sim\ (\nu z)(E[q] \mid q\,\mathsf{b}\,(\kappa, [\,]))$$

Proof. We start the proof with $E[\mathsf{newQSem}\ \kappa]$. Replacing $\mathsf{newQSem}$ by its implementation:

$$E[(\lambda k.\mathsf{let}\ l = \mathsf{Nil}; b = \textbf{newBuf}\ \mathsf{unit}\ \ \mathsf{in}\ \ \textbf{put}\ (b, (k, l)); b)\kappa]$$

$\xrightarrow{\beta\text{-CBV(ev)}}$

$$E[\mathsf{let}\ l = \mathsf{Nil}; b = \textbf{newBuf}\ \mathsf{unit}\ \ \mathsf{in}\ \ \textbf{put}\ (b, (\kappa, l)); b]$$

Removing let and in :

$$E[(\lambda\ l.(\lambda b.\textbf{put}\ (b, (\kappa, l)); b)\textbf{newBuf}\ \mathsf{unit}\ )\mathsf{Nil}]$$

$\xrightarrow{\beta\text{-CBV(ev)}}$

$$E[\lambda b.\textbf{put}\ (b, (\kappa, [\,])); b)\textbf{newBuf}\ \mathsf{unit}]$$

Note that Nil is represented as $[\,]$. We perform some reduction rules:

$$\begin{aligned}
&\xrightarrow{\text{BUFF.NEW(ev)}} && (\nu q)(E[(\lambda b.\textbf{put}\ (b, (\kappa, [\,])); b)q] \mid q\,\mathsf{b} -) \\
&\xrightarrow{\beta\text{-CBV(ev)}} && (\nu q)(E[\textbf{put}\ (q, (\kappa, [\,])); q] \mid q\,\mathsf{b} -) \\
&\xrightarrow{\text{DET.PUT}} && (\nu q)(E[\mathsf{unit}; q] \mid q\,\mathsf{b}\,(\kappa, [\,])) \\
&\to && (\nu q)(E[(\lambda\_.q)\mathsf{unit}] \mid q\,\mathsf{b}\,(\kappa, [\,])) \\
&\xrightarrow{\beta\text{-CBV(ev)}} && (\nu q)(E[q] \mid q\,\mathsf{b}\,(\kappa, [\,]))
\end{aligned}$$

The result is exactly the term that is also produced by the transformation rule TQSEM.NEW. □

Proposition 3.3.2.

$$(1)\quad (\nu q)(E[\mathit{down}\ q] \mid q\,\mathsf{b}\,(1,[\,])) \sim (\nu q)(E[\mathsf{True}] \mid q\,\mathsf{b}\,(0,[\,]))$$

$$\begin{aligned}(2)\quad & (\nu q)(E[\mathit{down}\ q] \mid q\,\mathsf{b}\,(0,[\,])) \\ & \sim (\nu q)(\nu h)(\nu f')(E[\mathit{wait}\ h] \mid q\,\mathsf{b}\,(0, h:[\,]) \mid h\,\mathsf{h}\,f')\end{aligned}$$

Proof.

Case 3.3.2.1. (1) We start directly with

$$(\nu q)(E[\mathsf{down}\ q] \mid q\,\mathsf{b}\,(1,[\,]))$$

Replacing down by its implementation (handle-version):

$$\begin{array}{c}(\nu q)(E[(\lambda qs.\mathsf{let}\ \langle cnt, ls\rangle = \mathbf{get}\ qs \\ \mathsf{in}\ \ \mathbf{case}\ (cnt == 0)\ \mathbf{of} \\ \mathsf{True} \rightarrow \mathsf{let}\ \langle x, f\rangle = \mathsf{newhandled} \\ \quad \mathsf{in}\ \ \mathbf{put}\ qs\ (cnt, x : ls); \mathbf{case}\ x\ \mathbf{of} \\ \qquad\qquad \mathsf{True} \rightarrow\ \mathsf{True} \\ \qquad\qquad \mathsf{False} \rightarrow\ \bot \\ \mathsf{False} \rightarrow\ \mathbf{put}\ qs\ (cnt-1, ls); \mathsf{True})\ q] \mid q\,\mathsf{b}\,(1,[\,]))\end{array}$$

We replace qs by q in the λ-expression using β-CBV(ev):

$$\begin{array}{c}(\nu q)(E[\mathsf{let}\ \langle cnt, ls\rangle = \mathbf{get}\ q \\ \mathsf{in}\ \ \mathbf{case}\ (cnt == 0)\ \mathbf{of} \\ \mathsf{True} \rightarrow \mathsf{let}\ \langle x, f\rangle = \mathsf{newhandled} \\ \quad \mathsf{in}\ \ \mathbf{put}\ q\ (cnt, x : ls); \mathbf{case}\ x\ \mathbf{of} \\ \qquad\qquad \mathsf{True} \rightarrow\ \mathsf{True} \\ \qquad\qquad \mathsf{False} \rightarrow\ \bot \\ \mathsf{False} \rightarrow\ \mathbf{put}\ q\ (cnt-1, ls); \mathsf{True}] \mid q\,\mathsf{b}\,(1,[\,]))\end{array}$$

Removing $\mathsf{let} \ldots\ \mathsf{in}$:

$$\begin{array}{c}(\nu q)(E[(\lambda\langle cnt, ls\rangle.\mathbf{case}\ (cnt == 0)\ \mathbf{of} \\ \mathsf{True} \rightarrow \mathsf{let}\ \langle x, f\rangle = \mathsf{newhandled} \\ \quad \mathsf{in}\ \ \mathbf{put}\ q\ (cnt, x : ls); \mathbf{case}\ x\ \mathbf{of} \\ \qquad\qquad \mathsf{True} \rightarrow\ \mathsf{True} \\ \qquad\qquad \mathsf{False} \rightarrow\ \bot \\ \mathsf{False} \rightarrow\ \mathbf{put}\ q\ (cnt-1, ls); \mathsf{True})\ \mathbf{get}\ q] \mid q\,\mathsf{b}\,(1,[\,]))\end{array}$$

Getting the contents of the quantity semaphore q with DET.GET leads to:

$$\begin{array}{c}(\nu q)(E[(\lambda\langle cnt, ls\rangle.\textbf{case}\ (cnt == 0)\ \textbf{of}\\ \mathsf{True} \to \mathsf{let}\ \langle x, f\rangle = \mathsf{newhandled}\\ \quad\mathsf{in}\ \ \textbf{put}\ q\ (cnt, x : ls); \textbf{case}\ x\ \textbf{of}\\ \qquad\qquad \mathsf{True} \to\ \mathsf{True}\\ \qquad\qquad \mathsf{False} \to\ \bot\\ \mathsf{False} \to\ \textbf{put}\ q\ (cnt - 1, ls); \mathsf{True})\ (1, [\,])] \mid q\,\mathsf{b}\,-)\end{array}$$

We β-CBV(ev)-reduce $(1, [\,])$ and bind it to $\langle cnt, ls\rangle$:

$$\begin{array}{c}(\nu q)(E[\textbf{case}\ (1 == 0)\ \textbf{of}\\ \mathsf{True} \to \mathsf{let}\ \langle x, f\rangle = \mathsf{newhandled}\\ \quad\mathsf{in}\ \ \textbf{put}\ q\ (1, x : [\,]); \textbf{case}\ x\ \textbf{of}\\ \qquad\qquad \mathsf{True} \to\ \mathsf{True}\\ \qquad\qquad \mathsf{False} \to\ \bot\\ \mathsf{False} \to\ \textbf{put}\ q\ (1 - 1, [\,]); \mathsf{True}] \mid q\,\mathsf{b}\,-)\end{array}$$

CASE.BETA(ev) reduces the term to the right-handside of the False-alternative:

$$(\nu q)(E[\textbf{put}\ q\ (1 - 1, [\,]); \mathsf{True}] \mid q\,\mathsf{b}\,-)$$

$\xrightarrow{\text{DET.PUT}}$

$$(\nu q)(E[\mathsf{unit}; \mathsf{True}] \mid q\,\mathsf{b}\,(0, [\,]))$$

We remove the sequential computation:

$$(\nu q)(E[(\lambda\_.\mathsf{True})\mathsf{unit}] \mid q\,\mathsf{b}\,(0, [\,]))$$

At last we β-CBV(ev)-remove unit:

$$(\nu q)(E[\mathsf{True}] \mid q\,\mathsf{b}\,(0, [\,]))$$

This is also what TDOWN in the case of a non-*zero*-valued quantity semaphore produces. Now we consider the other case:

Case 3.3.2.2. (2)

$$(\nu q)(E[\mathsf{down}\ q] \mid q\,\mathsf{b}\,(0, [\,]))$$

Replacing down by its implementation (handle-version):

$$\begin{array}{c}(\nu q)(E[(\lambda qs.\mathsf{let}\ \langle cnt, ls\rangle = \mathbf{get}\ qs\\ \mathsf{in}\ \ \mathbf{case}\ (cnt == 0)\ \mathbf{of}\\ \mathsf{True} \rightarrow \mathsf{let}\ \langle x, f\rangle = \mathsf{newhandled}\\ \mathsf{in}\ \ \mathbf{put}\ qs\ (cnt, x : ls); \mathbf{case}\ x\ \mathbf{of}\\ \mathsf{True} \rightarrow\ \mathsf{True}\\ \mathsf{False} \rightarrow\ \bot\\ \mathsf{False} \rightarrow\ \mathbf{put}\ qs\ (cnt - 1, ls); \mathsf{True})\ q]\ |\ q\,\mathsf{b}\,(0, [\,]))\end{array}$$

β-CBV(ev) replaces qs by q in the λ-expression:

$$\begin{array}{c}(\nu q)(E[\mathsf{let}\ \langle cnt, ls\rangle = \mathbf{get}\ q\\ \mathsf{in}\ \ \mathbf{case}\ (cnt == 0)\ \mathbf{of}\\ \mathsf{True} \rightarrow \mathsf{let}\ \langle x, f\rangle = \mathsf{newhandled}\\ \mathsf{in}\ \ \mathbf{put}\ q\ (cnt, x : ls); \mathbf{case}\ x\ \mathbf{of}\\ \mathsf{True} \rightarrow\ \mathsf{True}\\ \mathsf{False} \rightarrow\ \bot\\ \mathsf{False} \rightarrow\ \mathbf{put}\ q\ (cnt - 1, ls); \mathsf{True}]\ |\ q\,\mathsf{b}\,(0, [\,]))\end{array}$$

Removing $\mathsf{let} \ldots\ \mathsf{in}$:

$$\begin{array}{c}(\nu q)(E[(\lambda\langle cnt, ls\rangle.\mathbf{case}\ (cnt == 0)\ \mathbf{of}\\ \mathsf{True} \rightarrow \mathsf{let}\ \langle x, f\rangle = \mathsf{newhandled}\\ \mathsf{in}\ \ \mathbf{put}\ q\ (cnt, x : ls); \mathbf{case}\ x\ \mathbf{of}\\ \mathsf{True} \rightarrow\ \mathsf{True}\\ \mathsf{False} \rightarrow\ \bot\\ \mathsf{False} \rightarrow\ \mathbf{put}\ q\ (cnt - 1, ls); \mathsf{True})\ \mathbf{get}\ q]\ |\ q\,\mathsf{b}\,(0, [\,]))\end{array}$$

Getting the content of q with DET.GET:

$$\begin{array}{c}(\nu q)(E[(\lambda\langle cnt, ls\rangle.\mathbf{case}\ (cnt == 0)\ \mathbf{of}\\ \mathsf{True} \rightarrow \mathsf{let}\ \langle x, f\rangle = \mathsf{newhandled}\\ \mathsf{in}\ \ \mathbf{put}\ q\ (cnt, x : ls); \mathbf{case}\ x\ \mathbf{of}\\ \mathsf{True} \rightarrow\ \mathsf{True}\\ \mathsf{False} \rightarrow\ \bot\\ \mathsf{False} \rightarrow\ \mathbf{put}\ q\ (cnt - 1, ls); \mathsf{True})\ (0, [\,])]\ |\ q\,\mathsf{b}\,-)\end{array}$$

We replace $\langle cnt, ls\rangle$ via $(0, [\,])$ with a β-CBV(ev) reduction:

$$\begin{array}{l}(\nu q)(E[\textbf{case}\ (0 == 0)\ \textbf{of}\\ \quad \mathsf{True} \rightarrow \mathsf{let}\ \langle x, f\rangle = \mathsf{newhandled}\\ \quad\quad \mathsf{in}\ \ \textbf{put}\ q\ (0, x : [\,]); \textbf{case}\ x\ \textbf{of}\\ \quad\quad\quad \mathsf{True} \rightarrow\ \mathsf{True}\\ \quad\quad\quad \mathsf{False} \rightarrow\ \bot\\ \quad \mathsf{False} \rightarrow\ \textbf{put}\ q\ (0-1, [\,]); \mathsf{True}]\ |\ q\, \mathsf{b}\, -)\end{array}$$

Now the case-expression can be reduced with CASE.BETA(ev) to:

$$\begin{array}{l}(\nu q)(E[\mathsf{let}\ \langle x, f\rangle = \mathsf{newhandled}\ \ \mathsf{in}\ \ \textbf{put}\ q\ (0, x : [\,]); \textbf{case}\ x\ \textbf{of}\\ \quad \mathsf{True} \rightarrow\ \mathsf{True}\\ \quad \mathsf{False} \rightarrow\ \bot]\ |\ q\, \mathsf{b}\, -)\end{array}$$

Removing $\mathsf{let} \ldots\ \mathsf{in}$ results in:

$$\begin{array}{l}(\nu q)(E[(\lambda\langle x, f\rangle.\textbf{put}\ q\ (0, x : [\,]); \textbf{case}\ x\ \textbf{of}\\ \quad \mathsf{True} \rightarrow\ \mathsf{True}\\ \quad \mathsf{False} \rightarrow\ \bot)\mathsf{newhandled}]\ |\ q\, \mathsf{b}\, -)\end{array}$$

Creating new handle components with HANDLE.NEW$(ev) \rightarrow$

$$\begin{array}{l}(\nu q)(\nu h)(\nu f')(E[(\lambda\langle x, f\rangle.\textbf{put}\ q\ (0, x : [\,]); \textbf{case}\ x\ \textbf{of}\\ \quad \mathsf{True} \rightarrow\ \mathsf{True}\\ \quad \mathsf{False} \rightarrow\ \bot)\langle h, f'\rangle]\ |\ q\, \mathsf{b}\ -\ \ |\ h\, \mathsf{h}\, f')\end{array}$$

Beta-reducing the handle components $\xrightarrow{\beta\text{-CBV(ev)}}$

$$\begin{array}{l}(\nu q)(\nu h)(\nu f')(E[\textbf{put}\ q\ (0, h : [\,]); \textbf{case}\ h\ \textbf{of}\\ \quad \mathsf{True} \rightarrow\ \mathsf{True}\\ \quad \mathsf{False} \rightarrow\ \bot]\ |\ q\, \mathsf{b}\ -\ \ |\ h\, \mathsf{h}\, f')\end{array}$$

Putting in a new value to the quantity semaphore by DET.PUT:

$$\begin{array}{l}(\nu q)(\nu h)(\nu f')(E[\mathsf{unit}; \textbf{case}\ h\ \textbf{of}\\ \quad \mathsf{True} \rightarrow\ \mathsf{True}\\ \quad \mathsf{False} \rightarrow\ \bot]\ |\ q\, \mathsf{b}\, (0, h : [\,])\ |\ h\, \mathsf{h}\, f')\end{array}$$

Removing sequential computation results in:

$$\begin{array}{l}(\nu q)(\nu h)(\nu f')(E[(\lambda\_.\textbf{case}\ h\ \textbf{of}\\ \quad \mathsf{True} \rightarrow\ \mathsf{True}\\ \quad \mathsf{False} \rightarrow\ \bot)\mathsf{unit}]\ |\ q\, \mathsf{b}\, (0, h : [\,])\ |\ h\, \mathsf{h}\, f')\end{array}$$

Now we can bind unit to $\_$ with β-CBV(ev):

$$\begin{aligned}(\nu q)(\nu h)(\nu f')(E[&\textbf{case } h \textbf{ of}\\ &\quad\mathsf{True} \rightarrow \mathsf{True}\\ &\quad\mathsf{False} \rightarrow \bot] \mid q\,\mathsf{b}\,(0, h : [\,]) \mid h\,\mathsf{h}\,f')\end{aligned}$$

Finally, we can replace the case expression by shorthand wait:

$$(\nu q)(\nu h)(\nu f')(E[\mathsf{wait}\ h] \mid q\,\mathsf{b}\,(0, h : [\,]) \mid h\,\mathsf{h}\,f')$$

Since this is also the result of TDOWN in the case of a *zero*-valued quantity semaphore, we showed correctness of the rule TDOWN.

□

In the case of up we also must consider two cases.

Proposition 3.3.3.

(1) $(\nu q)(E[\mathsf{up}\ q] \mid q\,\mathsf{b}\,(0, [\,])) \sim (\nu q)(E[\mathsf{unit}] \mid q\,\mathsf{b}\,(1, [\,]))$

(2) $(\nu q)(\nu h)(\nu f)(E[\mathsf{up}\ q] \mid q\,\mathsf{b}\,(0, [h]) \mid h\,\mathsf{h}\,f)$
$\sim (\nu q)(\nu h)(\nu f)(E[\mathsf{unit}] \mid q\,\mathsf{b}\,(0, [\,]) \mid h\,\mathsf{h}\,\bullet \mid f \Leftarrow \mathsf{True})$

Proof.

Case 3.3.3.1. We start with the following program:

$$(\nu q)(E[\mathsf{up}\ q] \mid q\,\mathsf{b}\,(0, [\,]))$$

Replacing up by its implementation (handle-version):

$$\begin{aligned}(\nu q)(E[(\lambda qs.\\ &\mathsf{let}\ \langle cnt, ls\rangle = \textbf{get}\ qs\ \ \mathsf{in}\\ &\textbf{case}\ ls\ \textbf{of}\\ &[\,] \rightarrow \textbf{put}\ qs\ (cnt+1, ls)\\ &x : xs \rightarrow x\ \mathsf{True};\\ &\quad\textbf{put}\ qs\ (cnt, xs))\ q] \mid q\,\mathsf{b}\,(0, [\,]))\end{aligned}$$

β-CBV(ev) of q to qs:

$$\begin{aligned}(\nu q)(E[&\mathsf{let}\ \langle cnt, ls\rangle = \textbf{get}\ q\ \ \mathsf{in}\\ &\textbf{case}\ ls\ \textbf{of}\\ &[\,] \rightarrow \textbf{put}\ q\ (cnt+1, ls)\\ &x : xs \rightarrow x\ \mathsf{True};\\ &\quad\textbf{put}\ q\ (cnt, xs)] \mid q\,\mathsf{b}\,(0, [\,]))\end{aligned}$$

Removing $\mathsf{let} \ldots\ \mathsf{in}$:

$$\begin{array}{c}(\nu q)(E[(\lambda\langle cnt, ls\rangle. \\ \textbf{case}\ ls\ \textbf{of} \\ [\,] \to \textbf{put}\ q\ (cnt+1, ls) \\ x : xs \to\ x\ \mathsf{True}; \\ \textbf{put}\ q\ (cnt, xs))\ \textbf{get}\ q] \mid q\,\mathsf{b}\,(0, [\,]))\end{array}$$

Performing DET.GET on the quantity semaphore q:

$$\begin{array}{c}(\nu q)(E[(\lambda\langle cnt, ls\rangle. \\ \textbf{case}\ ls\ \textbf{of} \\ [\,] \to \textbf{put}\ q\ (cnt+1, ls) \\ x : xs \to\ x\ \mathsf{True}; \\ \textbf{put}\ q\ (cnt, xs))\ (0, [\,])] \mid q\,\mathsf{b}\,-)\end{array}$$

We bind $(0, [\,])$ to $\langle cnt, ls\rangle$ in the λ-expression with a β-CBV(ev)-reduction:

$$\begin{array}{c}(\nu q)(E[\textbf{case}\ [\,]\ \textbf{of} \\ [\,] \to \textbf{put}\ q\ (0+1, [\,]) \\ x : xs \to\ x\ \mathsf{True}; \\ \textbf{put}\ q\ (0, xs)] \mid q\,\mathsf{b}\,-)\end{array}$$

CASE.BETA(ev) evaluates the term to the first alternative's right-handside:

$$(\nu q)(E[\textbf{put}\ q\ (0+1, [\,])] \mid q\,\mathsf{b}\,-)$$

Putting $(0+1, [\,])$ into q with DET.PUT:

$$(\nu q)(E[\mathsf{unit}] \mid q\,\mathsf{b}\,(1, [\,]))$$

In this first case we can conclude that the corresponding case of the transformation rule TUP is correct. We only used correct reduction (resp. transformation) rules here.

Case 3.3.3.2. We consider the second case by starting with:

$$(\nu q)(\nu h)(\nu f)(E[\mathsf{up}\ q] \mid q\,\mathsf{b}\,(0, [h]) \mid h\,\mathsf{h}\,f)$$

Then, replacing up by its implementation (handle-version) results in:

$$\begin{array}{c}(\nu q)(\nu h)(\nu f)(E[(\lambda qs. \\ \mathsf{let}\ \langle cnt, ls\rangle = \textbf{get}\ qs\ \ \mathsf{in} \\ \textbf{case}\ ls\ \textbf{of} \\ [\,] \to \textbf{put}\ qs\ (cnt+1, ls) \\ x : xs \to\ x\ \mathsf{True}; \\ \textbf{put}\ qs\ (cnt, xs))\ q] \mid q\,\mathsf{b}\,(0, [h]) \mid h\,\mathsf{h}\,f)\end{array}$$

Replacing qs by q in the λ-expression with β-CBV(ev):

$$\begin{gathered}(\nu q)(\nu h)(\nu f)(E[\mathsf{let}\ \langle cnt, ls\rangle = \mathbf{get}\ q\ \ \mathsf{in}\\ \mathbf{case}\ ls\ \mathbf{of}\\ [\,] \to \mathbf{put}\ q\ (cnt+1, ls)\\ x : xs \to\ x\ \mathsf{True};\\ \mathbf{put}\ q\ (cnt, xs)] \mid q\, \mathsf{b}\, (0, [h]) \mid h\, \mathsf{h}\, f)\end{gathered}$$

We remove the $\mathsf{let} \ldots\ \mathsf{in}$ construct:

$$\begin{gathered}(\nu q)(\nu h)(\nu f)(E[(\lambda\langle cnt, ls\rangle.\\ \mathbf{case}\ ls\ \mathbf{of}\\ [\,] \to \mathbf{put}\ q\ (cnt+1, ls)\\ x : xs \to\ x\ \mathsf{True};\\ \mathbf{put}\ q\ (cnt, xs))\ \mathbf{get}\ q] \mid q\, \mathsf{b}\, (0, [h]) \mid h\, \mathsf{h}\, f)\end{gathered}$$

We get the contents of $q \xrightarrow{\text{DET.GET}}$

$$\begin{gathered}(\nu q)(\nu h)(\nu f)(E[(\lambda\langle cnt, ls\rangle.\\ \mathbf{case}\ ls\ \mathbf{of}\\ [\,] \to \mathbf{put}\ q\ (cnt+1, ls)\\ x : xs \to\ x\ \mathsf{True};\\ \mathbf{put}\ q\ (cnt, xs))\ (0, [h])] \mid q\, \mathsf{b}\ - \ \mid h\, \mathsf{h}\, f)\end{gathered}$$

Binding $(0, [h])$ to $\langle cnt, ls\rangle$ in the λ-expression via a β-CBV(ev)-reduction:

$$\begin{gathered}(\nu q)(\nu h)(\nu f)(E[\mathbf{case}\ [h]\ \mathbf{of}\\ [\,] \to \mathbf{put}\ q\ (0+1, [\,])\\ x : xs \to\ x\ \mathsf{True};\\ \mathbf{put}\ q\ (0, xs)] \mid q\, \mathsf{b}\ - \ \mid h\, \mathsf{h}\, f)\end{gathered}$$

Using a CASE.BETA(ev)-reduction on $[h]$ results in:

$$(\nu q)(\nu h)(\nu f)(E[h\ \mathsf{True}; \mathbf{put}\ q\ (0, [\,])] \mid q\, \mathsf{b}\ - \ \mid h\, \mathsf{h}\, f)$$

Removing sequential computation:

$$(\nu q)(\nu h)(\nu f)(E[(\lambda\_.\mathbf{put}\ q\ (0, [\,]))(h\ \mathsf{True})] \mid q\, \mathsf{b}\ - \ \mid h\, \mathsf{h}\, f)$$

We bind the handle h to True with HANDLE.BIND(ev):

$$(\nu q)(\nu h)(\nu f)(E[(\lambda\_.\mathbf{put}\ q\ (0, [\,]))\mathsf{unit}] \mid q\, \mathsf{b}\ - \ \mid h\, \mathsf{h}\ \bullet\ \mid f \Leftarrow \mathsf{True})$$

The last step produced a unit-value. We bind it to _ with a β-CBV(ev)-reduction:

$$(\nu q)(\nu h)(\nu f)(E[\textbf{put}\ q\ (0, [\,])] \mid q\,\mathsf{b}\ - \mid h\,\mathsf{h}\ \bullet \mid f {\Leftarrow} \mathsf{True})$$

Now we put new contents to the quantity semaphore $\xrightarrow{\text{DET.PUT}}$

$$(\nu q)(\nu h)(\nu f)(E[\mathsf{unit}] \mid q\,\mathsf{b}\,(0, [\,]) \mid h\,\mathsf{h}\ \bullet \mid f {\Leftarrow} \mathsf{True})$$

GC may remove the used handle and future:

$$(\nu z)(\nu q)(E[\mathsf{unit}] \mid q\,\mathsf{b}\,(0, [\,]))$$

With these two results which are exactly the results of TDOWN in both cases, we showed that TDOWN is a correct transformation rule. ☐

Theorem 4 (Correctness of transformation rules of the quantity semaphore abstraction in λ^{τ}(fchb))**.** *TQSEM.NEW, TUP and TDOWN are correct transformation rules in λ^{τ}(fchb).*

Proof. This follows from proposition 3.3.1, 3.3.3 and 3.3.2. ☐

3.4 Properties Of The Bounded Channel Abstraction

We show correct transformations for the bounded channel abstraction in λ^{τ}(fchb). Doing this we follow the same approach as for channels.

Proposition 3.4.1.

$$\begin{gathered} E[\mathit{newBChan}\ \kappa] \ \sim \\ (\nu z)(\nu q)(\nu c)(\nu r)(\nu w)(\nu h_1) \\ (E[(c, q)] \mid q\,\mathsf{b}\,(\kappa, [\,]) \mid c {\Leftarrow} (r, w) \mid r\,\mathsf{b}\,h_1 \mid h_1\,\mathsf{b}\ - \mid w\,\mathsf{b}\,h_1) \end{gathered}$$

Proof. We start with $E[\mathsf{newBChan}\ \kappa]$. Replacing the operation newBChan by its implementation results in:

$$E[(\lambda k.\mathsf{let}\ qsem = (\mathsf{newQSem}\ k), chan = (\mathsf{newChan}\ \mathsf{unit})\ \mathsf{in}\ \ (chan, qsem))\kappa]$$

$\xrightarrow{\beta\text{-CBV}(\mathsf{ev})}$

$$E[\mathsf{let}\ qsem = (\mathsf{newQSem}\ \kappa), chan = (\mathsf{newChan\ unit})\ \mathsf{in}\ (chan, qsem)]$$

Removing the let ... in construct:

$$E[(\lambda\ qsem.(\lambda\ chan.(chan, qsem))(\mathsf{newChan\ unit}))(\mathsf{newQSem}\ \kappa)]$$

We use TQSEM.NEW to create q. Since it is a correct transformation rule our reduction stays correct for now:

$$(\nu q)(E[(\lambda\ qsem.(\lambda\ chan.(chan, qsem))(\mathsf{newChan\ unit}))\ q] \mid q\,\mathsf{b}\,(\kappa, [\,]))$$

We replace $qsem$ with q in the λ-expression with β-CBV(ev):

$$(\nu q)(E[(\lambda\ chan.(chan, q))(\mathsf{newChan\ unit})] \mid q\,\mathsf{b}\,(\kappa, [\,]))$$

We use TCHAN.NEW to create a new channel. Since TCHAN.NEW is a correct transformation rule our reduction stays also correct for now:

$$\begin{gathered}(\nu q)(\nu c)(\nu r)(\nu w)(\nu h_1)(E[(\lambda\ chan.(chan, q))\ c] \\ \mid q\,\mathsf{b}\,(\kappa, [\,]) \mid c \Leftarrow (r, w) \mid r\,\mathsf{b}\,h_1 \mid h_1\,\mathsf{b}\,- \mid w\,\mathsf{b}\,h_1)\end{gathered}$$

Replacing $chan$ by c with β-CBV(ev):

$$\begin{gathered}(\nu q)(\nu c)(\nu r)(\nu w)(\nu h_1) \\ (E[(c, q)] \mid q\,\mathsf{b}\,(\kappa, [\,]) \mid c \Leftarrow (r, w) \mid r\,\mathsf{b}\,h_1 \mid h_1\,\mathsf{b}\,- \mid w\,\mathsf{b}\,h_1)\end{gathered}$$

Since we only used correct rules doing this, we can conclude that this is a correct transformation as well. □

Proposition 3.4.2.

$$\begin{gathered}(\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_1 \ldots b_{n+1}) \\ (E[\mathit{writeBChan}\ (c, v)] \mid c \Leftarrow (c', q) \mid q\,\mathsf{b}\,(\kappa - n, [\,]) \mid c' \Leftarrow (r, w) \\ \mid r\,\mathsf{b}\,b_1 \mid b_1\,\mathsf{b}\,\mathit{Item}(v_1, b_2) \\ \mid \ldots \mid b_n\,\mathsf{b}\,\mathit{Item}(v_n, b_{n+1}) \mid b_{n+1}\,\mathsf{b}\,- \mid w\,\mathsf{b}\,b_{n+1}) \\ \sim \\ (\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_1 \ldots b_{n+2}) \\ (E[\mathit{unit}] \mid c \Leftarrow (c', q) \mid q\,\mathsf{b}\,(\kappa - n - 1, [\,]) \mid c' \Leftarrow (r, w) \\ \mid r\,\mathsf{b}\,b_1 \mid b_1\,\mathsf{b}\,\mathit{Item}(v_1, b_2) \\ \mid \ldots \mid b_n\,\mathsf{b}\,\mathit{Item}(v_n, b_{n+1}) \mid b_{n+1}\,\mathsf{b}\,\mathit{Item}(v, b_{n+2}) \mid b_{n+2}\,\mathsf{b}\,- \mid w\,\mathsf{b}\,b_{n+2})\end{gathered}$$

$$for\ 0 < n < \kappa$$

Proof. We begin with the situation of a bounded channel that has already stored n values:

$$\begin{array}{c}
(\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_1 \dots b_{n+1}) \\
(E[\mathsf{writeBChan}\ (c, v)] \mid c \Leftarrow (c', q) \mid q\, \mathsf{b}\, (\kappa - n, [\,]) \mid c' \Leftarrow (r, w) \\
\mid r\, \mathsf{b}\, b_1 \mid b_1\, \mathsf{b}\, \mathsf{Item}(v_1, b_2) \mid \dots \mid b_n\, \mathsf{b}\, \mathsf{Item}(v_n, b_{n+1}) \mid b_{n+1}\, \mathsf{b}\, - \mid w\, \mathsf{b}\, b_{n+1})
\end{array}$$

Replacing $\mathsf{writeBChan}$ by its implementation leads to:

$$\begin{array}{c}
(\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_1 \dots b_{n+1}) \\
(E[(\lambda \langle c', qsem \rangle, v'\_. \\
\mathsf{down}\ qsem; \\
\mathsf{writeChan}\ c'\ v')(c, v)] \\
\mid c \Leftarrow (c', q) \mid q\, \mathsf{b}\, (\kappa - n, [\,]) \mid c' \Leftarrow (r, w) \\
\mid r\, \mathsf{b}\, b_1 \mid b_1\, \mathsf{b}\, \mathsf{Item}(v_1, b_2) \mid \dots \mid b_n\, \mathsf{b}\, \mathsf{Item}(v_n, b_{n+1}) \mid b_{n+1}\, \mathsf{b}\, - \mid w\, \mathsf{b}\, b_{n+1})
\end{array}$$

β-CBV(ev) reduction binds c to $\langle c', qsem \rangle$ and v to v':

$$\begin{array}{c}
(\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_1 \dots b_{n+1}) \\
(E[\mathsf{down}\ q; \mathsf{writeChan}\ c\ v] \\
\mid c \Leftarrow (c', q) \mid q\, \mathsf{b}\, (\kappa - n, [\,]) \mid c' \Leftarrow (r, w) \\
\mid r\, \mathsf{b}\, b_1 \mid b_1\, \mathsf{b}\, \mathsf{Item}(v_1, b_2) \mid \dots \mid b_n\, \mathsf{b}\, \mathsf{Item}(v_n, b_{n+1}) \mid b_{n+1}\, \mathsf{b}\, - \mid w\, \mathsf{b}\, b_{n+1})
\end{array}$$

Removing sequential computation:

$$\begin{array}{c}
(\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_1 \dots b_{n+1}) \\
(E[(\lambda\_.\mathsf{writeChan}\ c\ v)\mathsf{down}\ q] \\
\mid c \Leftarrow (c', q) \mid q\, \mathsf{b}\, (\kappa - n, [\,]) \mid c' \Leftarrow (r, w) \\
\mid r\, \mathsf{b}\, b_1 \mid b_1\, \mathsf{b}\, \mathsf{Item}(v_1, b_2) \mid \dots \mid b_n\, \mathsf{b}\, \mathsf{Item}(v_n, b_{n+1}) \mid b_{n+1}\, \mathsf{b}\, - \mid w\, \mathsf{b}\, b_{n+1})
\end{array}$$

We decrement the value of the quantity semaphore q using the correct transformation rule TDOWN:

$$\begin{array}{c}
(\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_1 \dots b_{n+1}) \\
(E[(\lambda\_.\mathsf{writeChan}\ c\ v)\mathsf{True}] \\
\mid c \Leftarrow (c', q) \mid q\, \mathsf{b}\, (\kappa - n - 1, [\,]) \mid c' \Leftarrow (r, w) \\
\mid r\, \mathsf{b}\, b_1 \mid b_1\, \mathsf{b}\, \mathsf{Item}(v_1, b_2) \mid \dots \mid b_n\, \mathsf{b}\, \mathsf{Item}(v_n, b_{n+1}) \mid b_{n+1}\, \mathsf{b}\, - \mid w\, \mathsf{b}\, b_{n+1})
\end{array}$$

Now a β-CBV(ev)-reduction on the term removes True:

$$\begin{array}{c}
(\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_1 \dots b_{n+1}) \\
(E[\mathsf{writeChan}\ c\ v] \\
\mid c \Leftarrow (c', q) \mid q\, \mathsf{b}\, (\kappa - n - 1, [\,]) \mid c' \Leftarrow (r, w) \\
\mid r\, \mathsf{b}\, b_1 \mid b_1\, \mathsf{b}\, \mathsf{Item}(v_1, b_2) \mid \dots \mid b_n\, \mathsf{b}\, \mathsf{Item}(v_n, b_{n+1}) \mid b_{n+1}\, \mathsf{b}\, - \mid w\, \mathsf{b}\, b_{n+1})
\end{array}$$

We use the correct transformation rule TCHAN.WRITE to write value v to the channel c:

$$\begin{gathered}(\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_1 \dots b_{n+1}, b_{n+2})\\ (E[\mathsf{unit}]\\ \mid c \Leftarrow (c', q) \mid q \,\mathsf{b}\, (\kappa - n - 1, [\,]) \mid c' \Leftarrow (r, w)\\ \mid r \,\mathsf{b}\, b_1 \mid b_1 \,\mathsf{b}\, \mathsf{Item}(v_1, b_2) \mid \dots \mid b_n \,\mathsf{b}\, \mathsf{Item}(v_n, b_{n+1}) \mid b_{n+1} \,\mathsf{b}\, \mathsf{Item}(v, b_{n+2})\\ \mid b_{n+2} \,\mathsf{b}\, - \mid w \,\mathsf{b}\, b_{n+2})\end{gathered}$$

With that we have proven the statement. □

Proposition 3.4.3.

$$\begin{gathered}(\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_1 \dots b_{n+1})\\ (E[\mathit{readBChan}\ c] \mid c \Leftarrow (c', q) \mid q \,\mathsf{b}\, (\kappa - n, [\,]) \mid c' \Leftarrow (r, w)\\ \mid r \,\mathsf{b}\, b_1 \mid b_1 \,\mathsf{b}\, \mathit{Item}(v_1, b_2) \mid \dots \mid b_n \,\mathsf{b}\, \mathit{Item}(v_n, b_{n+1}) \mid b_{n+1} \,\mathsf{b}\, - \mid w \,\mathsf{b}\, b_{n+1})\\ \sim\\ (\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_2 \dots b_{n+1})\\ (E[v_1] \mid c \Leftarrow (c', q) \mid q \,\mathsf{b}\, (\kappa - n + 1, [\,]) \mid c' \Leftarrow (r, w)\\ \mid r \,\mathsf{b}\, b_2 \mid b_2 \,\mathsf{b}\, \mathit{Item}(v_2, b_3) \mid \dots \mid b_n \,\mathsf{b}\, \mathit{Item}(v_n, b_{n+1}) \mid b_{n+1} \,\mathsf{b}\, - \mid w \,\mathsf{b}\, b_{n+1})\end{gathered}$$

$$for\ 0 < n \leq \kappa$$

Proof. We start with the following program:

$$\begin{gathered}(\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_1 \dots b_{n+1})\\ (E[\mathsf{readBChan}\ c] \mid c \Leftarrow (c', q) \mid q \,\mathsf{b}\, (\kappa - n, [\,]) \mid c' \Leftarrow (r, w)\\ \mid r \,\mathsf{b}\, b_1 \mid b_1 \,\mathsf{b}\, \mathsf{Item}(v_1, b_2) \mid \dots \mid b_n \,\mathsf{b}\, \mathsf{Item}(v_n, b_{n+1}) \mid b_{n+1} \,\mathsf{b}\, - \mid w \,\mathsf{b}\, b_{n+1})\end{gathered}$$

Replacing readBChan by its implementation:

$$\begin{gathered}(\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_1 \dots b_{n+1})\\ (E[(\lambda\langle c*, qsem\rangle.\mathsf{up}\ qsem; \mathsf{readChan}\ c*)\ c]\\ \mid c \Leftarrow (c', q) \mid q \,\mathsf{b}\, (\kappa - n, [\,]) \mid c' \Leftarrow (r, w)\\ \mid r \,\mathsf{b}\, b_1 \mid b_1 \,\mathsf{b}\, \mathsf{Item}(v_1, b_2) \mid \dots \mid b_n \,\mathsf{b}\, \mathsf{Item}(v_n, b_{n+1}) \mid b_{n+1} \,\mathsf{b}\, - \mid w \,\mathsf{b}\, b_{n+1})\end{gathered}$$

We dereference the bounded channel c's value (c', q):

$$\begin{gathered}(\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_1 \dots b_{n+1})\\ (E[(\lambda\langle c*, qsem\rangle.\mathsf{up}\ qsem; \mathsf{readChan}\ c*)\ (c', q)]\\ \mid c \Leftarrow (c', q) \mid q \,\mathsf{b}\, (\kappa - n, [\,]) \mid c' \Leftarrow (r, w)\\ \mid r \,\mathsf{b}\, b_1 \mid b_1 \,\mathsf{b}\, \mathsf{Item}(v_1, b_2) \mid \dots \mid b_n \,\mathsf{b}\, \mathsf{Item}(v_n, b_{n+1}) \mid b_{n+1} \,\mathsf{b}\, - \mid w \,\mathsf{b}\, b_{n+1})\end{gathered}$$

Now a β-CBV(ev)-reduction replaces $\lambda\langle c*, qsem\rangle$ with (c', q):

$$\begin{array}{c}(\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_1 \ldots b_{n+1})\\ (E[\mathsf{up}\ q; \mathsf{readChan}\ c'] \mid c \Leftarrow (c', q) \mid q\,\mathsf{b}\,(\kappa - n, [\,]) \mid c' \Leftarrow (r, w)\\ \mid r\,\mathsf{b}\,b_1 \mid b_1\,\mathsf{b}\,\mathsf{Item}(v_1, b_2) \mid \ \ldots\ \mid b_n\,\mathsf{b}\,\mathsf{Item}(v_n, b_{n+1}) \mid b_{n+1}\,\mathsf{b}\ - \mid w\,\mathsf{b}\,b_{n+1})\end{array}$$

We use the correct transformation rule TUP to increment the value of the quantity semaphore:

$$\begin{array}{c}(\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_1 \ldots b_{n+1})\\ (E[\mathsf{unit}; \mathsf{readChan}\ c'] \mid c \Leftarrow (c', q) \mid q\,\mathsf{b}\,(\kappa - n + 1, [\,]) \mid c' \Leftarrow (r, w)\\ \mid r\,\mathsf{b}\,b_1 \mid b_1\,\mathsf{b}\,\mathsf{Item}(v_1, b_2) \mid \ \ldots\ \mid b_n\,\mathsf{b}\,\mathsf{Item}(v_n, b_{n+1}) \mid b_{n+1}\,\mathsf{b}\ - \mid w\,\mathsf{b}\,b_{n+1})\end{array}$$

Removing sequential computation:

$$\begin{array}{c}(\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_1 \ldots b_{n+1})\\ (E[(\lambda\_.\mathsf{readChan}\ c')\mathsf{unit}] \mid c \Leftarrow (c', q) \mid q\,\mathsf{b}\,(\kappa - n + 1, [\,]) \mid c' \Leftarrow (r, w)\\ \mid r\,\mathsf{b}\,b_1 \mid b_1\,\mathsf{b}\,\mathsf{Item}(v_1, b_2) \mid \ \ldots\ \mid b_n\,\mathsf{b}\,\mathsf{Item}(v_n, b_{n+1}) \mid b_{n+1}\,\mathsf{b}\ - \mid w\,\mathsf{b}\,b_{n+1})\end{array}$$

$\xrightarrow{\beta\text{-CBV}(\mathsf{ev})}$

$$\begin{array}{c}(\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_1 \ldots b_{n+1})\\ (E[\mathsf{readChan}\ c'] \mid c \Leftarrow (c', q) \mid q\,\mathsf{b}\,(\kappa - n + 1, [\,]) \mid c' \Leftarrow (r, w)\\ \mid r\,\mathsf{b}\,b_1 \mid b_1\,\mathsf{b}\,\mathsf{Item}(v_1, b_2) \mid \ \ldots\ \mid b_n\,\mathsf{b}\,\mathsf{Item}(v_n, b_{n+1}) \mid b_{n+1}\,\mathsf{b}\ - \mid w\,\mathsf{b}\,b_{n+1})\end{array}$$

Now we perform the correct transformation rule TCHAN.READ to read a value from channel c':

$$\begin{array}{c}(\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_2 \ldots b_{n+1})\\ (E[v_1] \mid c \Leftarrow (c', q) \mid q\,\mathsf{b}\,(\kappa - n + 1, [\,]) \mid c' \Leftarrow (r, w)\\ \mid r\,\mathsf{b}\,b_2 \mid b_2\,\mathsf{b}\,\mathsf{Item}(v_2, b_3) \mid \ \ldots\ \mid b_n\,\mathsf{b}\,\mathsf{Item}(v_n, b_{n+1}) \mid b_{n+1}\,\mathsf{b}\ - \mid w\,\mathsf{b}\,b_{n+1})\end{array}$$

We have proven the proposition. $\square$

Proposition 3.4.4.

$$\begin{array}{c}(\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_1 \ldots b_{\kappa+1})(\nu h_1 \ldots h_m)(\nu f_1 \ldots f_m)\\ (E[\mathit{readBChan}\ c] \mid c \Leftarrow (c', q) \mid q\,\mathsf{b}\,(0, [h_1, \ldots, h_m]) \mid c' \Leftarrow (r, w)\\ \mid r\,\mathsf{b}\,b_1 \mid b_1\,\mathsf{b}\,\mathit{Item}(v_1, b_2) \mid \ \ldots\ \mid b_\kappa\,\mathsf{b}\,\mathit{Item}(v_\kappa, b_{\kappa+1}) \mid b_{\kappa+1}\,\mathsf{b}\,-\\ \mid w\,\mathsf{b}\,b_{\kappa+1} \mid h_1\,\mathsf{h}\,f_1 \mid \ \ldots\ \mid h_m\,\mathsf{h}\,f_m)\\ \sim\\ (\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_2 \ldots b_{\kappa+1})(\nu h_1 \ldots h_m)(\nu f_1 \ldots f_m)\\ (E[v_1] \mid c \Leftarrow (c', q) \mid q\,\mathsf{b}\,(0, [h_2, \ldots, h_m]) \mid c' \Leftarrow (r, w)\\ \mid r\,\mathsf{b}\,b_2 \mid b_2\,\mathsf{b}\,\mathit{Item}(v_2, b_3) \mid \ \ldots\ \mid b_\kappa\,\mathsf{b}\,\mathit{Item}(v_\kappa, b_{\kappa+1}) \mid b_{\kappa+1}\,\mathsf{b}\,-\\ \mid w\,\mathsf{b}\,b_{\kappa+1} \mid h_1\,\mathsf{h}\,\bullet \mid f_1 \Leftarrow \mathsf{True} \mid h_2\,\mathsf{h}\,f_2 \mid \ \ldots\ \mid h_m\,\mathsf{h}\,f_m)\end{array}$$

Proof. We start with:

$$\begin{gathered}(\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_1 \dots b_{\kappa+1})(\nu h_1 \dots h_m)(\nu f_1 \dots f_m)\\ (E[\mathsf{readBChan}\ c] \mid c \Leftarrow (c', q) \mid q\, \mathsf{b}\, (0, [h_1, \dots, h_m]) \mid c' \Leftarrow (r, w)\\ \mid r\, \mathsf{b}\, b_1 \mid b_1\, \mathsf{b}\, \mathsf{Item}(v_1, b_2) \mid \dots \mid b_\kappa\, \mathsf{b}\, \mathsf{Item}(v_\kappa, b_{\kappa+1}) \mid b_{\kappa+1}\, \mathsf{b} -\\ \mid w\, \mathsf{b}\, b_{\kappa+1} \mid h_1\, \mathsf{h}\, f_1 \mid \dots \mid h_m\, \mathsf{h}\, f_m)\end{gathered}$$

Replacing readBChan by its implementation:

$$\begin{gathered}(\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_1 \dots b_{\kappa+1})(\nu h_1 \dots h_m)(\nu f_1 \dots f_m)\\ (E[(\lambda\langle c*, qsem\rangle.\\ \mathsf{up}\ qsem;\\ \mathsf{readChan}\ c*)\ c] \mid c \Leftarrow (c', q) \mid q\, \mathsf{b}\, (0, [h_1, \dots, h_m]) \mid c' \Leftarrow (r, w)\\ \mid r\, \mathsf{b}\, b_1 \mid b_1\, \mathsf{b}\, \mathsf{Item}(v_1, b_2) \mid \dots \mid b_\kappa\, \mathsf{b}\, \mathsf{Item}(v_\kappa, b_{\kappa+1}) \mid b_{\kappa+1}\, \mathsf{b} -\\ \mid w\, \mathsf{b}\, b_{\kappa+1} \mid h_1\, \mathsf{h}\, f_1 \mid \dots \mid h_m\, \mathsf{h}\, f_m)\end{gathered}$$

We dereference the bounded channel c's value (c', q):

$$\begin{gathered}(\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_1 \dots b_{\kappa+1})(\nu h_1 \dots h_m)(\nu f_1 \dots f_m)\\ (E[(\lambda\langle c*, qsem\rangle.\\ \mathsf{up}\ qsem;\\ \mathsf{readChan}\ c*)\ (c', q)] \mid c \Leftarrow (c', q) \mid q\, \mathsf{b}\, (0, [h_1, \dots, h_m]) \mid c' \Leftarrow (r, w)\\ \mid r\, \mathsf{b}\, b_1 \mid b_1\, \mathsf{b}\, \mathsf{Item}(v_1, b_2) \mid \dots \mid b_\kappa\, \mathsf{b}\, \mathsf{Item}(v_\kappa, b_{\kappa+1}) \mid b_{\kappa+1}\, \mathsf{b} -\\ \mid w\, \mathsf{b}\, b_{\kappa+1} \mid h_1\, \mathsf{h}\, f_1 \mid \dots \mid h_m\, \mathsf{h}\, f_m)\end{gathered}$$

We β-CBV(ev)-reduce (c', q), binding it to $\langle c*, qsem\rangle$:

$$\begin{gathered}(\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_1 \dots b_{\kappa+1})(\nu h_1 \dots h_m)(\nu f_1 \dots f_m)\\ (E[\mathsf{up}\ q; \mathsf{readChan}\ c'] \mid c \Leftarrow (c', q) \mid q\, \mathsf{b}\, (0, [h_1, \dots, h_m]) \mid c' \Leftarrow (r, w)\\ \mid r\, \mathsf{b}\, b_1 \mid b_1\, \mathsf{b}\, \mathsf{Item}(v_1, b_2) \mid \dots \mid b_\kappa\, \mathsf{b}\, \mathsf{Item}(v_\kappa, b_{\kappa+1}) \mid b_{\kappa+1}\, \mathsf{b} -\\ \mid w\, \mathsf{b}\, b_{\kappa+1} \mid h_1\, \mathsf{h}\, f_1 \mid \dots \mid h_m\, \mathsf{h}\, f_m)\end{gathered}$$

Using the correct transformation rule TUP to increment the quantity semaphore's counter:

$$\begin{gathered}(\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_1 \dots b_{\kappa+1})(\nu h_1 \dots h_m)(\nu f_1 \dots f_m)\\ (E[\mathsf{unit}; \mathsf{readChan}\ c'] \mid c \Leftarrow (c', q) \mid q\, \mathsf{b}\, (0, [h_2, \dots, h_m]) \mid c' \Leftarrow (r, w)\\ \mid r\, \mathsf{b}\, b_1 \mid b_1\, \mathsf{b}\, \mathsf{Item}(v_1, b_2) \mid \dots \mid b_\kappa\, \mathsf{b}\, \mathsf{Item}(v_\kappa, b_{\kappa+1}) \mid b_{\kappa+1}\, \mathsf{b} -\\ \mid w\, \mathsf{b}\, b_{\kappa+1} \mid h_1\, \mathsf{h}\, \bullet \mid \dots \mid h_m\, \mathsf{h}\, f_m \mid f_1 \Leftarrow \mathsf{True})\end{gathered}$$

Removing sequential computation:

$$\begin{gathered}(\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_1 \ldots b_{\kappa+1})(\nu h_1 \ldots h_m)(\nu f_1 \ldots f_m)\\ (E[(\lambda\_.\mathsf{readChan}\ c')\mathsf{unit}] \mid c \Leftarrow (c', q) \mid q\,\mathsf{b}\,(0, [h_2, \ldots, h_m]) \mid c' \Leftarrow (r, w)\\ \mid r\,\mathsf{b}\,b_1 \mid b_1\,\mathsf{b}\,\mathsf{Item}(v_1, b_2) \mid \ldots \mid b_\kappa\,\mathsf{b}\,\mathsf{Item}(v_\kappa, b_{\kappa+1}) \mid b_{\kappa+1}\,\mathsf{b} -\\ \mid w\,\mathsf{b}\,b_{\kappa+1} \mid h_1\,\mathsf{h}\,\bullet \mid \ldots \mid h_m\,\mathsf{h}\,f_m \mid f_1 \Leftarrow \mathsf{True})\end{gathered}$$

$\xrightarrow{\beta\text{-CBV}(\mathsf{ev})}$

$$\begin{gathered}(\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_1 \ldots b_{\kappa+1})(\nu h_1 \ldots h_m)(\nu f_1 \ldots f_m)\\ (E[\mathsf{readChan}\ c'] \mid c \Leftarrow (c', q) \mid q\,\mathsf{b}\,(0, [h_2, \ldots, h_m]) \mid c' \Leftarrow (r, w)\\ \mid r\,\mathsf{b}\,b_1 \mid b_1\,\mathsf{b}\,\mathsf{Item}(v_1, b_2) \mid \ldots \mid b_\kappa\,\mathsf{b}\,\mathsf{Item}(v_\kappa, b_{\kappa+1}) \mid b_{\kappa+1}\,\mathsf{b} -\\ \mid w\,\mathsf{b}\,b_{\kappa+1} \mid h_1\,\mathsf{h}\,\bullet \mid \ldots \mid h_m\,\mathsf{h}\,f_m \mid f_1 \Leftarrow \mathsf{True})\end{gathered}$$

We read v_1 from the channel c' using the correct transformation rule TCHAN.READ:

$$\begin{gathered}(\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu b_2 \ldots b_{\kappa+1})(\nu h_1 \ldots h_m)(\nu f_1 \ldots f_m)\\ (E[v_1] \mid c \Leftarrow (c', q) \mid q\,\mathsf{b}\,(0, [h_2, \ldots, h_m]) \mid c' \Leftarrow (r, w)\\ \mid r\,\mathsf{b}\,b_2 \mid b_2\,\mathsf{b}\,\mathsf{Item}(v_2, b_3) \mid \ldots \mid b_\kappa\,\mathsf{b}\,\mathsf{Item}(v_\kappa, b_{\kappa+1}) \mid b_{\kappa+1}\,\mathsf{b} -\\ \mid w\,\mathsf{b}\,b_{\kappa+1} \mid h_1\,\mathsf{h}\,\bullet \mid \ldots \mid h_m\,\mathsf{h}\,f_m \mid f_1 \Leftarrow \mathsf{True})\end{gathered}$$

With that, we have proven the proposition. □

Theorem 5 (Correctness of transformation rules of the bounded channel abstraction in $\lambda^\tau(\mathsf{fchb})$)**.** TBCHAN.NEW*,* TBCHAN.WRITE *and* TBCHAN.READ *are correct transformation rules in* $\lambda^\tau(\mathit{fchb})$*.*

Proof. The correctness of TBCHAN.WRITE and TBCHAN.READ follows from proposition 3.4.2, 3.4.3 and 3.4.4. In proposition 3.4.1 we reduced the term to the same term that is produced by the rule TBCHAN.NEW after a FUT.DEREF and GC transformation:

$$\begin{gathered}(\nu q)(\nu c)(\nu r)(\nu w)(\nu h_1)(E[(c, q)]\\ \mid q\,\mathsf{b}\,(\kappa, [\,]) \mid c \Leftarrow (r, w) \mid r\,\mathsf{b}\,h_1 \mid h_1\,\mathsf{b}\ - \mid w\,\mathsf{b}\,h_1)\end{gathered}$$

$\xleftarrow{\text{GC}}$

$$\begin{gathered}(\nu z)(\nu q)(\nu c)(\nu r)(\nu w)(\nu h_1)(E[(c, q)] \mid z \Leftarrow (c, q)\\ \mid q\,\mathsf{b}\,(\kappa, [\,]) \mid c \Leftarrow (r, w) \mid r\,\mathsf{b}\,h_1 \mid h_1\,\mathsf{b}\ - \mid w\,\mathsf{b}\,h_1)\end{gathered}$$

$\xleftarrow{\text{FUT.DEREF}}$

$$\begin{gathered}(\nu z)(\nu q)(\nu c)(\nu r)(\nu w)(\nu h_1)(E[z] \mid z \Leftarrow (c, q)\\ \mid q\,\mathsf{b}\,(\kappa, [\,]) \mid c \Leftarrow (r, w) \mid r\,\mathsf{b}\,h_1 \mid h_1\,\mathsf{b}\ - \mid w\,\mathsf{b}\,h_1)\end{gathered}$$

Thus, TCHAN.NEW is also a correct transformation. □

3.5 Properties Of The Barrier Abstraction

We show correct transformations for the barrier abstraction in $\lambda^\tau(\mathsf{fchb})$. Doing so, we follow the same approach as for all the other abstractions.

Proposition 3.5.1.

$$E[\mathit{newBar}\ \kappa]\ \sim (\nu a)(\nu f)(E[(a, f, \kappa)] \mid a\,\mathsf{b}\,\kappa \mid f\,\mathsf{b}\,[\,])$$

Proof. We begin with $E[\mathsf{newBar}\ \kappa]$. Replacing newBar by its implementation:

$$\begin{gathered} E[(\lambda k.\mathsf{let}\ cnt_{act} = (\mathbf{newBuf}\ \mathsf{unit}),\\ cnt_{fin} = (\mathbf{newBuf}\ \mathsf{unit})\\ \mathsf{in}\ \mathbf{put}(cnt_{act}, k);\\ \mathbf{put}(cnt_{fin}, \mathsf{Nil});\\ (cnt_{act}, cnt_{fin}, k))\ \kappa] \end{gathered}$$

A β-CBV(ev)-reduction replaces k by κ in the λ-expression:

$$\begin{gathered} E[\mathsf{let}\ cnt_{act} = (\mathbf{newBuf}\ \mathsf{unit}),\\ cnt_{fin} = (\mathbf{newBuf}\ \mathsf{unit})\\ \mathsf{in}\ \mathbf{put}(cnt_{act}, \kappa);\\ \mathbf{put}(cnt_{fin}, \mathsf{Nil});\\ (cnt_{act}, cnt_{fin}, \kappa)] \end{gathered}$$

Replacing $\mathsf{let}\ \ldots\ \mathsf{in}$:

$$\begin{gathered} E[(\lambda\ cnt_{act}.(\lambda cnt_{fin}.\mathbf{put}(cnt_{act}, \kappa); \mathbf{put}(cnt_{fin}, \mathsf{Nil}); (cnt_{act}, cnt_{fin}, \kappa))\\ (\mathbf{newBuf}\ \mathsf{unit}))(\mathbf{newBuf}\ \mathsf{unit})] \end{gathered}$$

We create a new buffer a using BUFF.NEW(ev):

$$\begin{gathered} (\nu a)(E[(\lambda\ cnt_{act}.(\lambda cnt_{fin}.\mathbf{put}(cnt_{act}, \kappa); \mathbf{put}(cnt_{fin}, \mathsf{Nil}); (cnt_{act}, cnt_{fin}, \kappa))\\ (\mathbf{newBuf}\ \mathsf{unit}))a] \mid a\,\mathsf{b}\,-) \end{gathered}$$

A β-CBV(ev)-reduction replaces cnt_{act} by a:

$$\begin{gathered} (\nu a)(E[(\lambda cnt_{fin}.\mathbf{put}(a, \kappa); \mathbf{put}(cnt_{fin}, \mathsf{Nil}); (a, cnt_{fin}, \kappa))\\ (\mathbf{newBuf}\ \mathsf{unit})] \mid a\,\mathsf{b}\,-) \end{gathered}$$

Creating the second buffer f with BUFF.NEW(ev) $\rightarrow$

$$\begin{gathered} (\nu a)(\nu f)(E[(\lambda cnt_{fin}.\mathbf{put}(a, \kappa); \mathbf{put}(cnt_{fin}, \mathsf{Nil}); (a, cnt_{fin}, \kappa))f]\\ \mid a\,\mathsf{b}\,-\ \mid f\,\mathsf{b}\,-) \end{gathered}$$

A β-CBV(ev)-reduction replaces cnt_{fin} by f:

$$(\nu a)(\nu f)(E[\mathbf{put}(a,\kappa);\mathbf{put}(f,\mathsf{Nil});(a,f,\kappa)] \mid a\,\mathsf{b}\,- \mid f\,\mathsf{b}\,-)$$

Putting κ into buffer $a \xrightarrow{\text{DET.PUT}}$

$$(\nu a)(\nu f)(E[\mathsf{unit};\mathbf{put}(f,\mathsf{Nil});(a,f,\kappa)] \mid a\,\mathsf{b}\,\kappa \mid f\,\mathsf{b}\,-)$$

Removing sequential computation and β-CBV(ev)-reducing unit:

$$(\nu a)(\nu f)(E[\mathbf{put}(f,\mathsf{Nil});(a,f,\kappa)] \mid a\,\mathsf{b}\,\kappa \mid f\,\mathsf{b}\,-)$$

Putting the empty list Nil into buffer f with DET.PUT results in:

$$(\nu a)(\nu f)(E[\mathsf{unit};(a,f,\kappa)] \mid a\,\mathsf{b}\,\kappa \mid f\,\mathsf{b}\,[\,])$$

Removing sequential computation and β-CBV(ev) removes unit and results in:

$$(\nu a)(\nu f)(E[(a,f,\kappa)] \mid a\,\mathsf{b}\,\kappa \mid f\,\mathsf{b}\,[\,])$$

This transformation is correct since we only used correct rules. □

Proposition 3.5.2.

(1) $(\nu x)(\nu a)(\nu f)(\nu h_1 \ldots h_{\kappa-1})(\nu f_1 \ldots f_{\kappa-1})$
$(E[\mathit{syncBar}\ x] \mid x \Leftarrow (a,f,\kappa) \mid a\,\mathsf{b}\,1 \mid f\,\mathsf{b}\,[h_1 \ldots h_{\kappa-1}] \mid h_1\,\mathsf{h}\,f_1 \mid \ldots \mid h_{\kappa-1}\,\mathsf{h}\,f_{\kappa-1})$
$\sim\ (\nu x)(\nu a)(\nu f)(E[\mathsf{True}] \mid x \Leftarrow (a,f,\kappa) \mid a\,\mathsf{b}\,\kappa \mid f\,\mathsf{b}\,[\,])$

(2) $(\nu x)(\nu a)(\nu f)(\nu h_1 \ldots h_n)(\nu f_1 \ldots f_n)$
$(E[\mathit{syncBar}\ x] \mid x \Leftarrow (a,f,\kappa) \mid a\,\mathsf{b}\,\kappa - n \mid f\,\mathsf{b}\,[h_1 \ldots h_n] \mid h_1\,\mathsf{h}\,f_1 \mid \ldots \mid h_n\,\mathsf{h}\,f_n)$
$\sim\ (\nu x)(\nu a)(\nu f)(\nu h_1 \ldots h_{n+1})(\nu f_1 \ldots f_{n+1})$
$(E[h_{n+1}] \mid x \Leftarrow (a,f,\kappa) \mid a\,\mathsf{b}\,\kappa - n - 1 \mid f\,\mathsf{b}\,[h_1 \ldots h_{n+1}] \mid h_1\,\mathsf{h}\,f_1 \mid \ldots \mid h_{n+1}\,\mathsf{h}\,f_{n+1})$

Proof.

Case 3.5.2.1. We begin with:

$$\begin{gathered}(\nu x)(\nu f)(\nu a)(\nu h_1,\ldots,h_{\kappa-1})(\nu f_1,\ldots,f_{\kappa-1})\\ (E[\mathsf{syncBar}\ x] \mid x \Leftarrow (a,f,\kappa) \mid a\,\mathsf{b}\,1 \mid f\,\mathsf{b}\,[h_1 \ldots h_{\kappa-1}]\\ \mid h_1\,\mathsf{h}\,f_1 \mid \ldots \mid h_{\kappa-1}\,\mathsf{h}\,f_{\kappa-1})\end{gathered}$$

We replace $\mathsf{syncBar}$ by its implementation:

$$\begin{array}{c}
(\nu x)(\nu f)(\nu a)(\nu h_1,\ldots,h_{\kappa-1})(\nu f_1,\ldots,f_{\kappa-1}) \\
(E[(\lambda\langle cnt_{act}, cnt_{fin}, \kappa'\rangle. \\
\mathsf{let}\ act = (\mathbf{get}\ cnt_{act}), \\
\quad fin = (\mathbf{get}\ cnt_{fin}) \\
\mathsf{in}\ \ \mathbf{case}\ (act == 1)\ \mathbf{of} \\
\quad \mathsf{True} \rightarrow \mathbf{put}(cnt_{act}, \kappa'); \\
\mathbf{put}(cnt_{fin}, \mathsf{Nil}); \mathsf{openBar}\ fin; \mathsf{True} \\
\mathsf{False} \rightarrow \mathbf{put}(cnt_{act}, act - 1); \\
\quad \mathsf{let}\ \langle\ f, h\ \rangle = \mathsf{newhandled} \\
\mathsf{in}\ \ \mathbf{put}(cnt_{fin}, h : fin); h)\ x] \\
\mid x {\Leftarrow} (a, f, \kappa) \mid a\,\mathsf{b}\,1 \mid f\,\mathsf{b}\,[h_1 \ldots h_{\kappa-1}] \mid h_1\,\mathsf{h}\,f_1 \mid \ \ldots \ \mid h_{\kappa-1}\,\mathsf{h}\,f_{\kappa-1})
\end{array}$$

We dereference the value of x:

$$\begin{array}{c}
(\nu x)(\nu f)(\nu a)(\nu h_1,\ldots,h_{\kappa-1})(\nu f_1,\ldots,f_{\kappa-1}) \\
(E[(\lambda\langle cnt_{act}, cnt_{fin}, \kappa'\rangle. \\
\mathsf{let}\ act = (\mathbf{get}\ cnt_{act}), \\
\quad fin = (\mathbf{get}\ cnt_{fin}) \\
\mathsf{in}\ \ \mathbf{case}\ (act == 1)\ \mathbf{of} \\
\quad \mathsf{True} \rightarrow \mathbf{put}(cnt_{act}, \kappa'); \\
\mathbf{put}(cnt_{fin}, \mathsf{Nil}); \mathsf{openBar}\ fin; \mathsf{True} \\
\mathsf{False} \rightarrow \mathbf{put}(cnt_{act}, act - 1); \\
\quad \mathsf{let}\ \langle\ f, h\ \rangle = \mathsf{newhandled} \\
\mathsf{in}\ \ \mathbf{put}(cnt_{fin}, h : fin); h)\ (a, f, \kappa)] \\
\mid x {\Leftarrow} (a, f, \kappa) \mid a\,\mathsf{b}\,1 \mid f\,\mathsf{b}\,[h_1 \ldots h_{\kappa-1}] \mid h_1\,\mathsf{h}\,f_1 \mid \ \ldots \ \mid h_{\kappa-1}\,\mathsf{h}\,f_{\kappa-1})
\end{array}$$

Now we bind (a, f, κ) to $\langle cnt_{act}, cnt_{fin}, \kappa'\rangle$ with a β-$\textsc{cbv}(\mathsf{ev})$-reduction:

$$\begin{array}{c}
(\nu x)(\nu f)(\nu a)(\nu h_1,\ldots,h_{\kappa-1})(\nu f_1,\ldots,f_{\kappa-1}) \\
(E[\mathsf{let}\ act = (\mathbf{get}\ a), \\
\quad fin = (\mathbf{get}\ f) \\
\mathsf{in}\ \ \mathbf{case}\ (act == 1)\ \mathbf{of} \\
\mathsf{True} \rightarrow \mathbf{put}(a, \kappa); \mathbf{put}(f, \mathsf{Nil}); \mathsf{openBar}\ fin; \mathsf{True} \\
\mathsf{False} \rightarrow \mathbf{put}(a, act - 1); \\
\mathsf{let}\ \langle\ f, h\ \rangle = \mathsf{newhandled} \\
\mathsf{in}\ \ \mathbf{put}(f, h : fin); h] \\
\mid x {\Leftarrow} (a, f, \kappa) \mid a\,\mathsf{b}\,1 \mid f\,\mathsf{b}\,[h_1 \ldots h_{\kappa-1}] \mid h_1\,\mathsf{h}\,f_1 \mid \ \ldots \ \mid h_{\kappa-1}\,\mathsf{h}\,f_{\kappa-1})
\end{array}$$

Removing $\mathsf{let} \ldots \mathsf{in}$:

$$\begin{gathered}(\nu x)(\nu f)(\nu a)(\nu h_1, \ldots, h_{\kappa-1})(\nu f_1, \ldots, f_{\kappa-1}) \\ (E[(\lambda\ act.(\lambda fin.\textbf{case}\ (act == 1)\ \textbf{of} \\ \mathsf{True} \to \textbf{put}(a,\kappa); \textbf{put}(f, \mathsf{Nil}); \mathsf{openBar}\ fin; \mathsf{True} \\ \mathsf{False} \to \textbf{put}(a, act-1); \\ \mathsf{let}\ \langle\ f, h\ \rangle = \mathsf{newhandled} \\ \mathsf{in}\ \ \textbf{put}(f, h : fin); h)(\textbf{get}\ f))(\textbf{get}\ a)] \\ \mid x \Leftarrow (a, f, \kappa) \mid a\,\mathsf{b}\,1 \mid f\,\mathsf{b}\,[h_1 \ldots h_{\kappa-1}] \mid h_1\,\mathsf{h}\,f_1 \mid \ \ldots\ \mid h_{\kappa-1}\,\mathsf{h}\,f_{\kappa-1})\end{gathered}$$

Getting the contents of buffer a using DET.GET:

$$\begin{gathered}(\nu x)(\nu f)(\nu a)(\nu h_1, \ldots, h_{\kappa-1})(\nu f_1, \ldots, f_{\kappa-1}) \\ (E[(\lambda\ act.(\lambda fin.\textbf{case}\ (act == 1)\ \textbf{of} \\ \mathsf{True} \to \textbf{put}(a,\kappa); \textbf{put}(f, \mathsf{Nil}); \mathsf{openBar}\ fin; \mathsf{True} \\ \mathsf{False} \to \textbf{put}(a, act-1); \\ \mathsf{let}\ \langle\ f, h\ \rangle = \mathsf{newhandled} \\ \mathsf{in}\ \ \textbf{put}(f, h : fin); h)(\textbf{get}\ f))1] \\ \mid x \Leftarrow (a, f, \kappa) \mid a\,\mathsf{b}\,- \mid f\,\mathsf{b}\,[h_1 \ldots h_{\kappa-1}] \mid h_1\,\mathsf{h}\,f_1 \mid \ \ldots\ \mid h_{\kappa-1}\,\mathsf{h}\,f_{\kappa-1})\end{gathered}$$

Replacing act by 1 $\xrightarrow{\beta\text{-CBV}(\mathsf{ev})}$

$$\begin{gathered}(\nu x)(\nu f)(\nu a)(\nu h_1, \ldots, h_{\kappa-1})(\nu f_1, \ldots, f_{\kappa-1}) \\ (E[(\lambda fin.\textbf{case}\ (1 == 1)\ \textbf{of} \\ \mathsf{True} \to \textbf{put}(a,\kappa); \textbf{put}(f, \mathsf{Nil}); \mathsf{openBar}\ fin; \mathsf{True} \\ \mathsf{False} \to \textbf{put}(a, 1-1); \\ \mathsf{let}\ \langle\ f, h\ \rangle = \mathsf{newhandled} \\ \mathsf{in}\ \ \textbf{put}(f, h : fin); h)(\textbf{get}\ f)] \\ \mid x \Leftarrow (a, f, \kappa) \mid a\,\mathsf{b}\,- \mid f\,\mathsf{b}\,[h_1 \ldots h_{\kappa-1}] \mid h_1\,\mathsf{h}\,f_1 \mid \ \ldots\ \mid h_{\kappa-1}\,\mathsf{h}\,f_{\kappa-1})\end{gathered}$$

Getting the contents of f using DET.GET:

$$\begin{gathered}(\nu x)(\nu f)(\nu a)(\nu h_1, \ldots, h_{\kappa-1})(\nu f_1, \ldots, f_{\kappa-1}) \\ (E[(\lambda fin.\textbf{case}\ (1 == 1)\ \textbf{of} \\ \mathsf{True} \to \textbf{put}(a,\kappa); \textbf{put}(f, \mathsf{Nil}); \mathsf{openBar}\ fin; \mathsf{True} \\ \mathsf{False} \to \textbf{put}(a, 1-1); \\ \mathsf{let}\ \langle\ f, h\ \rangle = \mathsf{newhandled} \\ \mathsf{in}\ \ \textbf{put}(f, h : fin); h)[h_1 \ldots h_{\kappa-1}]] \\ \mid x \Leftarrow (a, f, \kappa) \mid a\,\mathsf{b}\,- \mid f\,\mathsf{b}\,- \mid h_1\,\mathsf{h}\,f_1 \mid \ \ldots\ \mid h_{\kappa-1}\,\mathsf{h}\,f_{\kappa-1})\end{gathered}$$

Binding the list of handles to fin in the λ-expression using β-CBV(ev):

$$\begin{gathered}(\nu x)(\nu f)(\nu a)(\nu h_1,\ldots,h_{\kappa-1})(\nu f_1,\ldots,f_{\kappa-1})\\ (E[\textbf{case}\ (1 == 1)\ \textbf{of}\\ \mathsf{True} \rightarrow \textbf{put}(a,\kappa); \textbf{put}(f,\mathsf{Nil}); \mathsf{openBar}\ [h_1 \ldots h_{\kappa-1}]; \mathsf{True}\\ \mathsf{False} \rightarrow \textbf{put}(a, 1-1);\\ \mathsf{let}\ \langle\ f, h\ \rangle = \mathsf{newhandled}\\ \mathsf{in}\ \ \textbf{put}(f, h : [h_1 \ldots h_{\kappa-1}]); h]\\ \mid x \Leftarrow (a,f,\kappa) \mid a\,\mathsf{b}\,- \mid f\,\mathsf{b}\,- \mid h_1\,\mathsf{h}\,f_1 \mid \ldots \mid h_{\kappa-1}\,\mathsf{h}\,f_{\kappa-1})\end{gathered}$$

Evaluating $(1 == 1)$:

$$\begin{gathered}(\nu x)(\nu f)(\nu a)(\nu h_1,\ldots,h_{\kappa-1})(\nu f_1,\ldots,f_{\kappa-1})\\ (E[\textbf{case}\ \mathsf{True}\ \textbf{of}\\ \mathsf{True} \rightarrow \textbf{put}(a,\kappa); \textbf{put}(f,\mathsf{Nil}); \mathsf{openBar}\ [h_1 \ldots h_{\kappa-1}]; \mathsf{True}\\ \mathsf{False} \rightarrow \textbf{put}(a, 1-1);\\ \mathsf{let}\ \langle\ f, h\ \rangle = \mathsf{newhandled}\\ \mathsf{in}\ \ \textbf{put}(f, h : [h_1 \ldots h_{\kappa-1}]); h]\\ \mid x \Leftarrow (a,f,\kappa) \mid a\,\mathsf{b}\,- \mid f\,\mathsf{b}\,- \mid h_1\,\mathsf{h}\,f_1 \mid \ldots \mid h_{\kappa-1}\,\mathsf{h}\,f_{\kappa-1})\end{gathered}$$

Reducing the case-expression with CASE.BETA(ev):

$$\begin{gathered}(\nu x)(\nu f)(\nu a)(\nu h_1,\ldots,h_{\kappa-1})(\nu f_1,\ldots,f_{\kappa-1})\\ (E[\textbf{put}(a,\kappa); \textbf{put}(f,\mathsf{Nil}); \mathsf{openBar}\ [h_1 \ldots h_{\kappa-1}]; \mathsf{True}]\\ \mid x \Leftarrow (a,f,\kappa) \mid a\,\mathsf{b}\,- \mid f\,\mathsf{b}\,- \mid h_1\,\mathsf{h}\,f_1 \mid \ldots \mid h_{\kappa-1}\,\mathsf{h}\,f_{\kappa-1})\end{gathered}$$

For improved readability we do not remove all the sequential computation here. We put κ into the buffer a using DET.PUT:

$$\begin{gathered}(\nu x)(\nu f)(\nu a)(\nu h_1,\ldots,h_{\kappa-1})(\nu f_1,\ldots,f_{\kappa-1})\\ (E[\mathsf{unit}; \textbf{put}(f,\mathsf{Nil}); \mathsf{openBar}\ [h_1 \ldots h_{\kappa-1}]; \mathsf{True}]\\ \mid x \Leftarrow (a,f,\kappa) \mid a\,\mathsf{b}\,\kappa \mid f\,\mathsf{b}\,- \mid h_1\,\mathsf{h}\,f_1 \mid \ldots \mid h_{\kappa-1}\,\mathsf{h}\,f_{\kappa-1})\end{gathered}$$

Removing sequential computation & GC removes unit:

$$\begin{gathered}(\nu x)(\nu f)(\nu a)(\nu h_1,\ldots,h_{\kappa-1})(\nu f_1,\ldots,f_{\kappa-1})\\ (E[\textbf{put}(f,\mathsf{Nil}); \mathsf{openBar}\ [h_1 \ldots h_{\kappa-1}]; \mathsf{True}]\\ \mid x \Leftarrow (a,f,\kappa) \mid a\,\mathsf{b}\,\kappa \mid f\,\mathsf{b}\,- \mid h_1\,\mathsf{h}\,f_1 \mid \ldots \mid h_{\kappa-1}\,\mathsf{h}\,f_{\kappa-1})\end{gathered}$$

Putting a new empty list Nil into buffer f with DET.PUT:

$$\begin{gathered}(\nu x)(\nu f)(\nu a)(\nu h_1,\ldots,h_{\kappa-1})(\nu f_1,\ldots,f_{\kappa-1})\\ (E[\mathsf{unit}; \mathsf{openBar}\ [h_1 \ldots h_{\kappa-1}]; \mathsf{True}] \mid x \Leftarrow (a,f,\kappa)\\ \mid a\,\mathsf{b}\,\kappa \mid f\,\mathsf{b}\,[\,] \mid h_1\,\mathsf{h}\,f_1 \mid \ldots \mid h_{\kappa-1}\,\mathsf{h}\,f_{\kappa-1})\end{gathered}$$

We remove sequential computation & garbage collect unit with GC:

$$(\nu x)(\nu f)(\nu a)(\nu h_1,\dots,h_{\kappa-1})(\nu f_1,\dots,f_{\kappa-1})$$
$$(E[\mathsf{openBar}\ [h_1\dots h_{\kappa-1}];\mathsf{True}]$$
$$\mid x\Leftarrow(a,f,\kappa)\mid a\,\mathsf{b}\,\kappa\mid f\,\mathsf{b}\,[\,]\mid h_1\,\mathsf{h}\,f_1\mid\ \dots\ \mid h_{\kappa-1}\,\mathsf{h}\,f_{\kappa-1})$$

We assume an internal reduction rule TOPEN.
$(\nu h_1,\dots,h_m)(\nu f_1,\dots,f_m)$
$(E[\mathsf{openBar}[h_1,\dots,h_m]]\mid f\,\mathsf{b}\,[h_1\dots h_m]\mid h_1\,\mathsf{h}\,f_1\mid\ \dots\ \mid h_m\,\mathsf{h}\,f_m)$
$\to(\nu h_1,\dots,h_m)(\nu f_1,\dots,f_m)(E[\mathsf{True}]\mid h_1\,\mathsf{h}\,\bullet\mid\ \dots\ \mid h_m\,\mathsf{h}\,\bullet\mid$
$f_1\Leftarrow\mathsf{True}\mid\ \dots\ \mid f_m\Leftarrow\mathsf{True})$ that shorts the reduction steps in $\mathsf{syncBar}$. TOPEN uses all the handles and binds the futures to True:

$$(\nu x)(\nu f)(\nu a)(\nu h_1,\dots,h_{\kappa-1})(\nu f_1,\dots,f_{\kappa-1})$$
$$(E[\mathsf{unit};\mathsf{True}]\mid x\Leftarrow(a,f,\kappa)\mid a\,\mathsf{b}\,\kappa\mid f\,\mathsf{b}\,[\,]\mid h_1\,\mathsf{h}\,\bullet\mid\ \dots\ \mid h_{\kappa-1}\,\mathsf{h}\,\bullet$$
$$\mid f_1\Leftarrow\mathsf{True}\mid\ \dots\ \mid f_{\kappa-1}\Leftarrow\mathsf{True})$$

Removing sequential computation:

$$(\nu x)(\nu f)(\nu a)(\nu h_1,\dots,h_{\kappa-1})(\nu f_1,\dots,f_{\kappa-1})$$
$$(E[(\lambda\_.\mathsf{True})\mathsf{unit}]\mid\ \mid x\Leftarrow(a,f,\kappa)\mid a\,\mathsf{b}\,\kappa\mid f\,\mathsf{b}\,[\,]\mid h_1\,\mathsf{h}\,\bullet\mid\ \dots\ \mid h_{\kappa-1}\,\mathsf{h}\,\bullet$$
$$\mid f_1\Leftarrow\mathsf{True}\mid\ \dots\ \mid f_{\kappa-1}\Leftarrow\mathsf{True})$$

Beta-reducing unit:

$$(\nu x)(\nu f)(\nu a)(\nu h_1,\dots,h_{\kappa-1})(\nu f_1,\dots,f_{\kappa-1})$$
$$(E[\mathsf{True}]\mid\ \mid x\Leftarrow(a,f,\kappa)\mid a\,\mathsf{b}\,\kappa\mid f\,\mathsf{b}\,[\,]\mid h_1\,\mathsf{h}\,\bullet\mid\ \dots\ \mid h_{\kappa-1}\,\mathsf{h}\,\bullet$$
$$\mid f_1\Leftarrow\mathsf{True}\mid\ \dots\ \mid f_{\kappa-1}\Leftarrow\mathsf{True})$$

Using GC removes all used handles and successful futures and results in:

$$(\nu x)(\nu f)(\nu a)(E[\mathsf{True}]\mid x\Leftarrow(a,f,\kappa)\mid a\,\mathsf{b}\,\kappa\mid f\,\mathsf{b}\,[\,])$$

Since we used TOPEN, of which we do not know if it is correct, we cannot conclude, that this reduction is correct.

Case 3.5.2.2. We begin with

$$(\nu x)(\nu f)(\nu a)(\nu h_1,\dots,h_n)(\nu f_1,\dots,f_n)$$
$$(E[\mathsf{syncBar}\ x]\mid x\Leftarrow(a,f,\kappa)\mid a\,\mathsf{b}\,\kappa-n\mid f\,\mathsf{b}\,[h_1\dots h_n]$$
$$\mid h_1\,\mathsf{h}\,f_1\mid\ \dots\ \mid h_n\,\mathsf{h}\,f_n)$$

Replacing syncBar by its implementation:

$$\begin{array}{c}
(\nu x)(\nu f)(\nu a)(\nu h_1,\ldots,h_n)(\nu f_1,\ldots,f_n)\\
(E[(\lambda\langle cnt_{act}, cnt_{fin}, \kappa'\rangle.\\
\mathsf{let}\ act = (\textbf{get}\ cnt_{act}),\\
fin = (\textbf{get}\ cnt_{fin})\\
\mathsf{in}\ \ \textbf{case}\ (act == 1)\ \textbf{of}\\
\mathsf{True} \rightarrow \textbf{put}(cnt_{act}, \kappa'); \textbf{put}(cnt_{fin}, \mathsf{Nil}); \mathsf{openBar}\ fin; \mathsf{True}\\
\mathsf{False} \rightarrow \textbf{put}(cnt_{act}, act - 1);\\
\mathsf{let}\ \langle\ f, h\ \rangle = \mathsf{newhandled}\\
\mathsf{in}\ \ \textbf{put}(cnt_{fin}, h : fin); h)x]\\
\mid x \Leftarrow (a, f, \kappa) \mid a\,\mathsf{b}\,\kappa - n \mid f\,\mathsf{b}\,[h_1 \ldots h_n]\\
\mid h_1\,\mathsf{h}\,f_1 \mid \ \ldots\ \mid h_n\,\mathsf{h}\,f_n)
\end{array}$$

Dereferencing the value of x leads to:

$$\begin{array}{c}
(\nu x)(\nu f)(\nu a)(\nu h_1,\ldots,h_n)(\nu f_1,\ldots,f_n)\\
(E[(\lambda\langle cnt_{act}, cnt_{fin}, \kappa'\rangle.\\
\mathsf{let}\ act = (\textbf{get}\ cnt_{act}),\\
fin = (\textbf{get}\ cnt_{fin})\\
\mathsf{in}\ \ \textbf{case}\ (act == 1)\ \textbf{of}\\
\mathsf{True} \rightarrow \textbf{put}(cnt_{act}, \kappa'); \textbf{put}(cnt_{fin}, \mathsf{Nil}); \mathsf{openBar}\ fin; \mathsf{True}\\
\mathsf{False} \rightarrow \textbf{put}(cnt_{act}, act - 1);\\
\mathsf{let}\ \langle\ f, h\ \rangle = \mathsf{newhandled}\\
\mathsf{in}\ \ \textbf{put}(cnt_{fin}, h : fin); h)(a, f, \kappa)]\\
\mid x \Leftarrow (a, f, \kappa) \mid a\,\mathsf{b}\,\kappa - n \mid f\,\mathsf{b}\,[h_1 \ldots h_n]\\
\mid h_1\,\mathsf{h}\,f_1 \mid \ \ldots\ \mid h_n\,\mathsf{h}\,f_n)
\end{array}$$

Renaming variable f in $\langle\ f, h\ \rangle$ to $f*$:

$$\begin{array}{c}
(\nu x)(\nu f)(\nu a)(\nu h_1,\ldots,h_n)(\nu f_1,\ldots,f_n)\\
(E[(\lambda\langle cnt_{act}, cnt_{fin}, \kappa'\rangle.\\
\mathsf{let}\ act = (\textbf{get}\ cnt_{act}),\\
fin = (\textbf{get}\ cnt_{fin})\\
\mathsf{in}\ \ \textbf{case}\ (act == 1)\ \textbf{of}\\
\mathsf{True} \rightarrow \textbf{put}(cnt_{act}, \kappa'); \textbf{put}(cnt_{fin}, \mathsf{Nil}); \mathsf{openBar}\ fin; \mathsf{True}\\
\mathsf{False} \rightarrow \textbf{put}(cnt_{act}, act - 1);\\
\mathsf{let}\ \langle\ f*, h\ \rangle = \mathsf{newhandled}\\
\mathsf{in}\ \ \textbf{put}(cnt_{fin}, h : fin); h)(a, f, \kappa)]\\
\mid x \Leftarrow (a, f, \kappa) \mid a\,\mathsf{b}\,\kappa - n \mid f\,\mathsf{b}\,[h_1 \ldots h_n]\\
\mid h_1\,\mathsf{h}\,f_1 \mid \ \ldots\ \mid h_n\,\mathsf{h}\,f_n)
\end{array}$$

Replacing $\langle cnt_{act}, cnt_{fin}, \kappa' \rangle$ with (a, f, κ) with β-CBV(ev):

$$\begin{gathered}
(\nu x)(\nu f)(\nu a)(\nu h_1, \ldots, h_n)(\nu f_1, \ldots, f_n) \\
(E[\mathsf{let}\ act = (\mathbf{get}\ a), \\
fin = (\mathbf{get}\ f) \\
\mathsf{in}\ \ \mathbf{case}\ (act == 1)\ \mathbf{of} \\
\mathsf{True} \rightarrow \mathbf{put}(a, \kappa); \mathbf{put}(f, \mathsf{Nil}); \mathsf{openBar}\ fin; \mathsf{True} \\
\mathsf{False} \rightarrow \mathbf{put}(a, act - 1); \\
\langle\ f*, h\ \rangle = \mathsf{newhandled} \\
\mathsf{in}\ \ \mathbf{put}(f, h : fin); h] \\
\mid x \Leftarrow (a, f, \kappa) \mid a\,\mathsf{b}\,\kappa - n \mid f\,\mathsf{b}\,[h_1 \ldots h_n] \\
\mid h_1\,\mathsf{h}\,f_1 \mid \ \ldots \ \mid h_n\,\mathsf{h}\,f_n)
\end{gathered}$$

Removing $\mathsf{let}\ \ldots\ \mathsf{in}$:

$$\begin{gathered}
(\nu x)(\nu f)(\nu a)(\nu h_1, \ldots, h_n)(\nu f_1, \ldots, f_n) \\
(E[(\lambda act.(\lambda fin.\mathbf{case}\ (act == 1)\ \mathbf{of} \\
\mathsf{True} \rightarrow \mathbf{put}(a, \kappa); \mathbf{put}(f, \mathsf{Nil}); \mathsf{openBar}\ fin; \mathsf{True} \\
\mathsf{False} \rightarrow \mathbf{put}(a, act - 1); \\
\mathsf{let}\ \langle\ f*, h\ \rangle = \mathsf{newhandled} \\
\mathsf{in}\ \ \mathbf{put}(f, h : fin); h)(\mathbf{get}\ f))(\mathbf{get}\ a)] \\
\mid x \Leftarrow (a, f, \kappa) \mid a\,\mathsf{b}\,\kappa - n \mid f\,\mathsf{b}\,[h_1 \ldots h_n] \\
\mid h_1\,\mathsf{h}\,f_1 \mid \ \ldots \ \mid h_n\,\mathsf{h}\,f_n)
\end{gathered}$$

Getting buffer a's values by DET.GET:

$$\begin{gathered}
(\nu x)(\nu f)(\nu a)(\nu h_1, \ldots, h_n)(\nu f_1, \ldots, f_n) \\
(E[(\lambda act.(\lambda fin.\mathbf{case}\ (act == 1)\ \mathbf{of} \\
\mathsf{True} \rightarrow \mathbf{put}(a, \kappa); \mathbf{put}(f, \mathsf{Nil}); \mathsf{openBar}\ fin; \mathsf{True} \\
\mathsf{False} \rightarrow \mathbf{put}(a, act - 1); \\
\mathsf{let}\ \langle\ f*, h\ \rangle = \mathsf{newhandled} \\
\mathsf{in}\ \ \mathbf{put}(f, h : fin); h)(\mathbf{get}\ f))\kappa - n] \\
\mid x \Leftarrow (a, f, \kappa) \mid a\,\mathsf{b}\ - \ \mid f\,\mathsf{b}\,[h_1 \ldots h_n] \\
\mid h_1\,\mathsf{h}\,f_1 \mid \ \ldots \ \mid h_n\,\mathsf{h}\,f_n)
\end{gathered}$$

Binding $\kappa - n$ to *act* in the λ-expression using a β-CBV(ev)-reduction:

$$\begin{array}{c}
(\nu x)(\nu f)(\nu a)(\nu h_1,\ldots,h_n)(\nu f_1,\ldots,f_n)\\
(E[(\lambda fin.\textbf{case}\ ((\kappa - n) == 1)\ \textbf{of}\\
\mathsf{True} \rightarrow \textbf{put}(a,\kappa);\textbf{put}(f,\mathsf{Nil});\mathsf{openBar}\ fin;\mathsf{True}\\
\mathsf{False} \rightarrow \textbf{put}(a,(\kappa - n) - 1);\\
\mathsf{let}\ \langle\ f*,h\ \rangle = \mathsf{newhandled}\\
\mathsf{in}\ \ \textbf{put}(f,h:fin);h)(\textbf{get}\ f)]\\
\mid x \Leftarrow (a,f,\kappa) \mid a\,\mathsf{b}\ - \ \mid f\,\mathsf{b}\,[h_1\ldots h_n]\\
\mid h_1\,\mathsf{h}\,f_1 \mid \ \ldots\ \mid h_n\,\mathsf{h}\,f_n)
\end{array}$$

Getting the contents of buffer $f \xrightarrow{\text{DET.GET}}$

$$\begin{array}{c}
(\nu x)(\nu f)(\nu a)(\nu h_1,\ldots,h_n)(\nu f_1,\ldots,f_n)\\
(E[(\lambda fin.\textbf{case}\ ((\kappa - n) == 1)\ \textbf{of}\\
\mathsf{True} \rightarrow \textbf{put}(a,\kappa);\textbf{put}(f,\mathsf{Nil});\mathsf{openBar}\ fin;\mathsf{True}\\
\mathsf{False} \rightarrow \textbf{put}(a,(\kappa - n) - 1);\\
\mathsf{let}\ \langle\ f*,h\ \rangle = \mathsf{newhandled}\\
\mathsf{in}\ \ \textbf{put}(f,h:fin);h)[h_1\ldots h_n]]\\
\mid x \Leftarrow (a,f,\kappa) \mid a\,\mathsf{b}\ - \ \mid f\,\mathsf{b}\,-\\
\mid h_1\,\mathsf{h}\,f_1 \mid \ \ldots\ \mid h_n\,\mathsf{h}\,f_n)
\end{array}$$

Binding the list of handles to $fin \xrightarrow{\beta\text{-CBV(ev)}}$

$$\begin{array}{c}
(\nu x)(\nu f)(\nu a)(\nu h_1,\ldots,h_n)(\nu f_1,\ldots,f_n)\\
(E[\textbf{case}\ ((\kappa - n) == 1)\ \textbf{of}\\
\mathsf{True} \rightarrow \textbf{put}(a,\kappa);\textbf{put}(f,\mathsf{Nil});\mathsf{openBar}\ [h_1\ldots h_n];\mathsf{True}\\
\mathsf{False} \rightarrow \textbf{put}(a,(\kappa - n) - 1);\\
\mathsf{let}\ \langle\ f*,h\ \rangle = \mathsf{newhandled}\\
\mathsf{in}\ \ \textbf{put}(f,h:[h_1\ldots h_n]);h]\\
\mid x \Leftarrow (a,f,\kappa) \mid a\,\mathsf{b}\ - \ \mid f\,\mathsf{b}\,-\\
\mid h_1\,\mathsf{h}\,f_1 \mid \ \ldots\ \mid h_n\,\mathsf{h}\,f_n)
\end{array}$$

Computing $((\kappa - n) == 1)$:

$$\begin{array}{c}(\nu x)(\nu f)(\nu a)(\nu h_1,\ldots,h_n)(\nu f_1,\ldots,f_n)\\ (E[\textbf{case}\ \mathsf{False}\ \textbf{of}\\ \mathsf{True} \to \textbf{put}(a,\kappa);\textbf{put}(f,\mathsf{Nil});\mathsf{openBar}\ [h_1\ldots h_n];\mathsf{True}\\ \mathsf{False} \to \textbf{put}(a,(\kappa-n)-1);\\ \mathsf{let}\ \langle\ f*,h\ \rangle = \mathsf{newhandled}\\ \mathsf{in}\ \ \textbf{put}(f,h:[h_1\ldots h_n]);h]\\ \mid x\Leftarrow(a,f,\kappa) \mid a\,\mathsf{b}\,- \mid f\,\mathsf{b}\,-\\ \mid h_1\,\mathsf{h}\,f_1 \mid \ldots \mid h_n\,\mathsf{h}\,f_n)\end{array}$$

Using CASE.BETA(ev) we enter the second alternative:

$$\begin{array}{c}(\nu x)(\nu f)(\nu a)(\nu h_1,\ldots,h_n)(\nu f_1,\ldots,f_n)\\ (E[\textbf{put}(a,(\kappa-n)-1);\\ \mathsf{let}\ \langle\ f*,h\ \rangle = \mathsf{newhandled}\\ \mathsf{in}\ \ \textbf{put}(f,h:[h_1\ldots h_n]);h]\\ \mid x\Leftarrow(a,f,\kappa) \mid a\,\mathsf{b}\,- \mid f\,\mathsf{b}\,-\\ \mid h_1\,\mathsf{h}\,f_1 \mid \ldots \mid h_n\,\mathsf{h}\,f_n)\end{array}$$

Putting $(\kappa - n) - 1$ into buffer a with DET.PUT:

$$\begin{array}{c}(\nu x)(\nu f)(\nu a)(\nu h_1,\ldots,h_n)(\nu f_1,\ldots,f_n)\\ (E[\mathsf{unit};\\ \mathsf{let}\ \langle\ f*,h\ \rangle = \mathsf{newhandled}\\ \mathsf{in}\ \ \textbf{put}(f,h:[h_1\ldots h_n]);h]\\ \mid x\Leftarrow(a,f,\kappa) \mid a\,\mathsf{b}\,\kappa-n-1 \mid f\,\mathsf{b}\,-\\ \mid h_1\,\mathsf{h}\,f_1 \mid \ldots \mid h_n\,\mathsf{h}\,f_n)\end{array}$$

Removing sequential computation and GC removes unit:

$$\begin{array}{c}(\nu x)(\nu f)(\nu a)(\nu h_1,\ldots,h_n)(\nu f_1,\ldots,f_n)\\ (E[\mathsf{let}\ \langle\ f*,h\ \rangle = \mathsf{newhandled}\\ \mathsf{in}\ \ \textbf{put}(f,h:[h_1\ldots h_n]);h]\\ \mid x\Leftarrow(a,f,\kappa) \mid a\,\mathsf{b}\,\kappa-n-1 \mid f\,\mathsf{b}\,-\\ \mid h_1\,\mathsf{h}\,f_1 \mid \ldots \mid h_n\,\mathsf{h}\,f_n)\end{array}$$

Replacing let ... in

$$\begin{array}{c}(\nu x)(\nu f)(\nu a)(\nu h_1,\ldots,h_n)(\nu f_1,\ldots,f_n)\\ (E[(\lambda\langle\ f*,h\ \rangle.\textbf{put}(f,h:[h_1\ldots h_n]);h)\mathsf{newhandled}]\\ \mid x\Leftarrow(a,f,\kappa) \mid a\,\mathsf{b}\,\kappa-n-1 \mid f\,\mathsf{b}\,-\\ \mid h_1\,\mathsf{h}\,f_1 \mid \ldots \mid h_n\,\mathsf{h}\,f_n)\end{array}$$

Creating a new handle $\langle\ f_{n+1}, h_{n+1}\ \rangle$ with HANDLE.NEW(ev):

$$\begin{gathered}(\nu x)(\nu f)(\nu a)(\nu h_1,\ldots,h_n,h_{n+1})(\nu f_1,\ldots,f_n,f_{n+1})\\ (E[(\lambda\langle\ f*,h\ \rangle.\mathbf{put}(f,h:[h_1\ldots h_n]);h)\langle\ f_{n+1},h_{n+1}\ \rangle]\\ \mid x\Leftarrow(a,f,\kappa)\mid a\,\mathsf{b}\,\kappa-n-1\mid f\,\mathsf{b}-\\ \mid h_1\,\mathsf{h}\,f_1\mid\ \ldots\ \mid h_n\,\mathsf{h}\,f_n\mid h_{n+1}\,\mathsf{h}\,f_{n+1})\end{gathered}$$

β-CBV(ev)-reducing the handle binds $\langle\ f_{n+1}, h_{n+1}\ \rangle$ to $\langle\ f*, h\ \rangle$:

$$\begin{gathered}(\nu x)(\nu f)(\nu a)(\nu h_1,\ldots,h_n,h_{n+1})(\nu f_1,\ldots,f_n,f_{n+1})\\ (E[\mathbf{put}(f,h_{n+1}:[h_1\ldots h_n]);h_{n+1}]\\ \mid x\Leftarrow(a,f,\kappa)\mid a\,\mathsf{b}\,\kappa-n-1\mid f\,\mathsf{b}-\\ \mid h_1\,\mathsf{h}\,f_1\mid\ \ldots\ \mid h_{n+1}\,\mathsf{h}\,f_{n+1})\end{gathered}$$

Putting the modified list $[h_1\ldots h_n, h_{n+1}]$ into buffer $f \xrightarrow{\text{DET.PUT}}$

$$\begin{gathered}(\nu x)(\nu f)(\nu a)(\nu h_1,\ldots,h_n,h_{n+1})(\nu f_1,\ldots,f_n,f_{n+1})\\ (E[\mathsf{unit};h_{n+1}]\mid x\Leftarrow(a,f,\kappa)\mid a\,\mathsf{b}\,\kappa-n-1\mid f\,\mathsf{b}\,[h_1\ldots h_n,h_{n+1}]\\ \mid h_1\,\mathsf{h}\,f_1\mid\ \ldots\ \mid h_{n+1}\,\mathsf{h}\,f_{n+1})\end{gathered}$$

Removing sequential computation and GC results in:

$$\begin{gathered}(\nu x)(\nu f)(\nu a)(\nu h_1,\ldots,h_n,h_{n+1})(\nu f_1,\ldots,f_n,f_{n+1})\\ (E[h_{n+1}]\mid x\Leftarrow(a,f,\kappa)\mid a\,\mathsf{b}\,\kappa-n-1\mid f\,\mathsf{b}\,[h_1\ldots h_n,h_{n+1}]\\ \mid h_1\,\mathsf{h}\,f_1\mid\ \ldots\ \mid h_{n+1}\,\mathsf{h}\,f_{n+1})\end{gathered}$$

□

Theorem 6 (Correctness of transformation rules of the barrier abstraction in $\lambda^\tau(\mathsf{fchb})$)**.** TBAR.NEW *is a correct transformation rule in* $\lambda^\tau(\mathsf{fchb})$*.*

Proof. We can reversely transform the result from proposition 3.5.1.

$$\begin{aligned}&(\nu a)(\nu f)(E[(a,f,\kappa)]\mid a\,\mathsf{b}\,\kappa\mid f\,\mathsf{b}\,[\,])\\ \xleftarrow{\text{GC}}\ &(\nu z)(\nu a)(\nu f)(E[(a,f,\kappa)]\mid z\Leftarrow(a,f,\kappa)\mid a\,\mathsf{b}\,\kappa\mid f\,\mathsf{b}\,[\,])\\ \xleftarrow{\text{FUT.DEREF}}\ &(\nu z)(\nu a)(\nu f)(E[z]\mid z\Leftarrow(a,f,\kappa)\mid a\,\mathsf{b}\,\kappa\mid f\,\mathsf{b}\,[\,])\end{aligned}$$

Thus, TBAR.NEW is a correct transformation rule. □

3.6 Program Equivalence

In this section we show equivalence of programs that use different concurrency abstractions.

Proposition 3.6.1.

$$\begin{array}{ll}
(1) & E[\textbf{thread}\ (\lambda\_.let\ x = (\mathit{newChan}\ \mathit{unit}) \\
 & in\ \mathit{writeChan}(x\ (v_1, v_2)); \mathit{readChan}(x); \mathit{readChan}(x); x)] \\
 & \sim \\
(2) & E[\mathit{newChan}\ \mathit{unit}]
\end{array}$$

Case 3.6.1.1. Reducing (1) by applying THREAD.NEW $\rightarrow$

$$(\nu z)(E[z] \mid z \Leftarrow (\lambda\_.\mathsf{let}\ x = (\mathsf{newChan\ unit}) \\ \mathsf{in}\ \ \mathsf{writeChan}(x\ (v_1, v_2)); \mathsf{readChan}(x); \mathsf{readChan}(x); x)z)$$

Beta-reducing $z \rightarrow$

$$(\nu z)(E[z] \mid z \Leftarrow \mathsf{let}\ x = (\mathsf{newChan\ unit}) \\ \mathsf{in}\ \ \mathsf{writeChan}(x\ (v_1, v_2)); \mathsf{readChan}(x); \mathsf{readChan}(x); x)$$

Removing syntactical let:

$$(\nu z)(E[z] \mid \\ z \Leftarrow (\lambda x.(\lambda\_.(\lambda\_.(\lambda\_.x)\mathsf{readChan}\ x)\mathsf{readChan}\ x)\mathsf{writeChan}(x\ (v_1, v_2)))\mathsf{newChan\ unit})$$

$\xrightarrow{\text{TCHAN.NEW}}$

$$(\nu z)(\nu c)(\nu r)(\nu w)(\nu h_1)(E[z] \mid \\ z \Leftarrow (\lambda x.(\lambda\_.(\lambda\_.(\lambda\_.x)\mathsf{readChan}\ x)\mathsf{readChan}\ x)\mathsf{writeChan}(x\ (v_1, v_2)))\ c \\ \mid c \Leftarrow (r, w) \mid r \mathsf{b}\, h_1 \mid h_1 \mathsf{b}\ - \ \mid w \mathsf{b}\, h_1)$$

$\xrightarrow{\beta\text{-CBV}(\mathsf{ev})}$

$$(\nu z)(\nu c)(\nu r)(\nu w)(\nu h_1)(E[z] \mid \\ z \Leftarrow (\lambda\_.(\lambda\_.(\lambda\_.c)\mathsf{readChan}\ c)\mathsf{readChan}\ c)\mathsf{writeChan}(x\ (v_1, v_2)) \\ \mid c \Leftarrow (r, w) \mid r \mathsf{b}\, h_1 \mid h_1 \mathsf{b}\ - \ \mid w \mathsf{b}\, h_1)$$

$\xrightarrow{\text{TCHAN.WRITEN}}$

$$(\nu z)(\nu c)(\nu r)(\nu w)(\nu h_1)(\nu h_2)(\nu h_3)(E[z] \mid \\ z \Leftarrow (\lambda\_.(\lambda\_.(\lambda\_.c)\mathsf{readChan}\ c)\mathsf{readChan}\ c)\mathsf{unit} \\ \mid c \Leftarrow (r, w) \mid r \mathsf{b}\, h_1 \mid h_1 \mathsf{b}\, \mathsf{Item}(v_1, h_2) \mid h_2 \mathsf{b}\, \mathsf{Item}(v_2, h_3) \mid h_3 \mathsf{b}\ - \ \mid w \mathsf{b}\, h_3)$$

$\xrightarrow{\beta\text{-CBV}(\mathsf{ev})}$

$$(\nu z)(\nu c)(\nu r)(\nu w)(\nu h_1)(\nu h_2)(\nu h_3)(E[z] \mid$$
$$z{\Leftarrow}(\lambda\_.(\lambda\_.c)\mathsf{readChan}\ c)\mathsf{readChan}\ c$$
$$\mid c \Leftarrow (r,w) \mid r\,\mathsf{b}\,h_1 \mid h_1\,\mathsf{b}\,\mathsf{Item}(v_1,h_2) \mid h_2\,\mathsf{b}\,\mathsf{Item}(v_2,h_3) \mid h_3\,\mathsf{b}\,- \mid w\,\mathsf{b}\,h_3)$$

$\xrightarrow{\text{TCHAN.READ}}$

$$(\nu z)(\nu c)(\nu r)(\nu w)(\nu h_1)(\nu h_2)(\nu h_3)(E[z] \mid$$
$$z{\Leftarrow}(\lambda\_.(\lambda\_.c)\mathsf{readChan}\ c)\ v_1$$
$$\mid c \Leftarrow (r,w) \mid r\,\mathsf{b}\,h_2 \mid h_1\,\mathsf{b}\,- \mid h_2\,\mathsf{b}\,\mathsf{Item}(v_2,h_3) \mid h_3\,\mathsf{b}\,- \mid w\,\mathsf{b}\,h_3)$$

$\xrightarrow{\text{GC}}$

$$(\nu z)(\nu c)(\nu r)(\nu w)(\nu h_2)(\nu h_3)(E[z] \mid$$
$$z{\Leftarrow}(\lambda\_.(\lambda\_.c)\mathsf{readChan}\ c)\ v_1$$
$$\mid c \Leftarrow (r,w) \mid r\,\mathsf{b}\,h_2 \mid h_2\,\mathsf{b}\,\mathsf{Item}(v_2,h_3) \mid h_3\,\mathsf{b}\,- \mid w\,\mathsf{b}\,h_3)$$

$\xrightarrow{\beta\text{-CBV}(\mathsf{ev})}$

$$(\nu z)(\nu c)(\nu r)(\nu w)(\nu h_2)(\nu h_3)(E[z] \mid$$
$$z{\Leftarrow}(\lambda\_.c)\mathsf{readChan}\ c$$
$$\mid c \Leftarrow (r,w) \mid r\,\mathsf{b}\,h_2 \mid h_2\,\mathsf{b}\,\mathsf{Item}(v_2,h_3) \mid h_3\,\mathsf{b}\,- \mid w\,\mathsf{b}\,h_3)$$

$\xrightarrow{\text{TCHAN.READ}}$

$$(\nu z)(\nu c)(\nu r)(\nu w)(\nu h_2)(\nu h_3)(E[z] \mid$$
$$z{\Leftarrow}(\lambda\_.c)\ v_2$$
$$\mid c \Leftarrow (r,w) \mid r\,\mathsf{b}\,h_3 \mid h_2\,\mathsf{b}\,- \mid h_3\,\mathsf{b}\,- \mid w\,\mathsf{b}\,h_3)$$

$\xrightarrow{\beta\text{-CBV}(\mathsf{ev})}$

$$(\nu z)(\nu c)(\nu r)(\nu w)(\nu h_2)(\nu h_3)(E[z] \mid z{\Leftarrow}c$$
$$\mid c \Leftarrow (r,w) \mid r\,\mathsf{b}\,h_3 \mid h_2\,\mathsf{b}\,- \mid h_3\,\mathsf{b}\,- \mid w\,\mathsf{b}\,h_3)$$

We dereference thread z's value with a FUT.DEREF:

$$(\nu z)(\nu c)(\nu r)(\nu w)(\nu h_2)(\nu h_3)(E[c] \mid z{\Leftarrow}c$$
$$\mid c \Leftarrow (r,w) \mid r\,\mathsf{b}\,h_3 \mid h_2\,\mathsf{b}\,- \mid h_3\,\mathsf{b}\,- \mid w\,\mathsf{b}\,h_3)$$

Now the garbage collection removes the empty buffer h_2 and successfully evaluated thread $z \rightarrow$

$$(\nu c)(\nu r)(\nu w)(\nu h_3)(E[c] \mid c \Leftarrow (r,w) \mid r\,\mathsf{b}\,h_3 \mid h_3\,\mathsf{b}\,- \mid w\,\mathsf{b}\,h_3)$$

The contexts evaluates to the empty channel.

Case 3.6.1.2. Reducing (2) with TCHAN.NEW

$$(\nu c)(\nu r)(\nu w)(\nu h_0)(E[c] \mid c \Leftarrow (r, w) \mid r\,\mathsf{b}\,h_0 \mid h_0\,\mathsf{b}\,- \mid w\,\mathsf{b}\,h_0)$$

The contexts evaluates to the empty channel.
The resulting process contains only the empty channel. (1) and (2) reduce to α-equivalent[1] processes.

Proposition 3.6.2.

$$\begin{array}{ll}(1) & E[\mathit{let}\ b = \mathit{newBar}\ 1\ \ \mathit{in}\ \ \mathit{syncBar}\ b] \\ & \sim \\ (2) & E[\mathit{let}\ q = \mathit{newQSem}\ n\ \ \mathit{in}\ \ \mathit{down}\ q]\end{array}$$

for $n > 0$, if TSYNC *was correct.*

Case 3.6.2.1. We start with **thread**$(\lambda\_.\mathsf{let}\ b = \mathsf{newBar}\ 1\ \ \mathsf{in}\ \ \mathsf{syncBar}\ b)$. Then THREAD.NEW results in:

$$(\nu z)(E[z] \mid z{\Leftarrow}(\lambda\_.\mathsf{let}\ b = \mathsf{newBar}\ 1\ \ \mathsf{in}\ \ \mathsf{syncBar}\ b)z)$$

$\xrightarrow{\beta\text{-CBV}(\mathsf{ev})}$

$$(\nu z)(E[z] \mid z{\Leftarrow}\mathsf{let}\ b = \mathsf{newBar}\ 1\ \ \mathsf{in}\ \ \mathsf{syncBar}\ b)$$

$\rightarrow$

$$(\nu z)(E[z] \mid z{\Leftarrow}(\lambda b.\mathsf{syncBar}\ b)\mathsf{newBar}\ 1)$$

$\xrightarrow{\text{BAR.NEW}(\mathsf{ev})}$

$$(\nu z)(\nu a)(\nu f)(\nu x)(E[z] \mid z{\Leftarrow}(\lambda b.\mathsf{syncBar}\ b)x \mid x{\Leftarrow}(a, f, 1) \mid a\,\mathsf{b}\,1 \mid f\,\mathsf{b}\,[\,])$$

$\xrightarrow{\beta\text{-CBV}(\mathsf{ev})}$

$$(\nu z)(\nu a)(\nu f)(\nu x)(E[z] \mid z{\Leftarrow}\mathsf{syncBar}\ x \mid x{\Leftarrow}(a, f, 1) \mid a\,\mathsf{b}\,1 \mid f\,\mathsf{b}\,[\,])$$

$\xrightarrow{\text{BAR.SYNC}(\mathsf{ev})}$

$$(\nu z)(\nu a)(\nu f)(\nu x)(E[z] \mid z{\Leftarrow}\mathsf{True} \mid x{\Leftarrow}(a, f, 1) \mid a\,\mathsf{b}\,1 \mid f\,\mathsf{b}\,[\,])$$

[1] α-equivalent terms are equal except for renaming.

$\xrightarrow{\text{FUT.DEREF(ev)}}$

$$(\nu z)(\nu a)(\nu f)(\nu x)(E[\mathsf{True}] \mid z \Leftarrow \mathsf{True} \mid x \Leftarrow (a, f, 1) \mid a\,\mathsf{b}\,1 \mid f\,\mathsf{b}\,[\,])$$

$\xrightarrow{\text{GC}}$

$$(\nu z)(\nu a)(\nu f)(\nu x)(E[\mathsf{True}] \mid x \Leftarrow (a, f, 1) \mid a\,\mathsf{b}\,1 \mid f\,\mathsf{b}\,[\,])$$

Case 3.6.2.2. We start with **thread**$(\lambda\_.\mathsf{let}\ q = \mathsf{newQSem}\ n\ \ \mathsf{in}\ \ \mathsf{down}\ q)$. Then THREAD.NEW results in:

$$(\nu z)(E[z] \mid z \Leftarrow (\lambda\_.\mathsf{let}\ q = \mathsf{newQSem}\ n\ \ \mathsf{in}\ \ \mathsf{down}\ q)z)$$

$\xrightarrow{\beta\text{-CBV(ev)}}$

$$(\nu z)(E[z] \mid z \Leftarrow \mathsf{let}\ q = \mathsf{newQSem}\ n\ \ \mathsf{in}\ \ \mathsf{down}\ q)$$

$\rightarrow$

$$(\nu z)(E[z] \mid z \Leftarrow (\lambda q.\mathsf{down}\ q)\mathsf{newQSem}\ n)$$

TQSEM.NEW $\rightarrow$

$$(\nu z)(\nu x)(E[z] \mid z \Leftarrow (\lambda q.\mathsf{down}\ q)x \mid x\,\mathsf{b}\,(n, [\,]))$$

$\xrightarrow{\beta\text{-CBV(ev)}}$

$$(\nu z)(\nu x)(E[z] \mid z \Leftarrow \mathsf{down}\ x \mid x\,\mathsf{b}\,(n, [\,]))$$

TDOWN $\rightarrow$

$$(\nu z)(\nu x)(E[z] \mid z \Leftarrow \mathsf{True} \mid x\,\mathsf{b}\,(n-1, [\,]))$$

FUT.DEREF(ev) $\rightarrow$

$$(\nu z)(\nu x)(E[\mathsf{True}] \mid z \Leftarrow \mathsf{True} \mid x\,\mathsf{b}\,(n-1, [\,]))$$

GC $\rightarrow$

$$(\nu x)(E[\mathsf{True}] \mid x\,\mathsf{b}\,(n-1, [\,]))$$

Proposition 3.6.3.

(1) $E[\mathit{let}\ b = \mathit{newBChan}\ 1\ \ \mathit{in}\ \mathit{writeBChan}\ (b, \mathsf{True}); \mathit{readBChan}\ b]$
$\sim$
(2) $E[\mathit{let}\ q = \mathit{newQSem}\ n\ \mathit{in}\ \mathit{down}\ q]$

for $n > 0$

Case 3.6.3.1. $\textbf{thread}(\lambda\_.\mathsf{let}\ b = \mathsf{newBChan}\ 1\ \mathsf{in}\ \mathsf{writeBChan}\ (b, \mathsf{True}); \mathsf{readBChan}\ b)$
THREAD.NEW $\longrightarrow$

$$(\nu z)(E[z] \mid z{\Leftarrow}(\lambda\_.\mathsf{let}\ b = \mathsf{newBChan}\ 1\ \ \mathsf{in}\ \ \mathsf{writeBChan}\ (b, \mathsf{True}); \mathsf{readBChan}\ b)z)$$

$\xrightarrow{\beta\text{-CBV}(\mathsf{ev})}$

$$(\nu z)(E[z] \mid z{\Leftarrow}\mathsf{let}\ b = \mathsf{newBChan}\ 1\ \ \mathsf{in}\ \ \mathsf{writeBChan}\ (b, \mathsf{True}); \mathsf{readBChan}\ b)$$

$\longrightarrow$

$$(\nu z)(E[z] \mid z{\Leftarrow}(\lambda b.\mathsf{writeBChan}\ (b, \mathsf{True}); \mathsf{readBChan}\ b)\mathsf{newBChan}\ 1)$$

TBCHAN.NEW $\longrightarrow$

$$\begin{gathered}(\nu z)(\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu h_0)\\ (E[z] \mid z{\Leftarrow}(\lambda b.\mathsf{writeBChan}\ (b, \mathsf{True}); \mathsf{readBChan}\ b)c \mid c{\Leftarrow}(c', q) \mid q\,\mathsf{b}\,(1, [\,])\\ \mid c'{\Leftarrow}(r, w) \mid r\,\mathsf{b}\,h_0 \mid h_0\,\mathsf{b}\ - \ \mid w\,\mathsf{b}\,h_0)\end{gathered}$$

$\xrightarrow{\beta\text{-CBV}(\mathsf{ev})}$

$$\begin{gathered}(\nu z)(\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu h_0)\\ (E[z] \mid z{\Leftarrow}\mathsf{writeBChan}\ (c, \mathsf{True}); \mathsf{readBChan}\ c \mid c{\Leftarrow}(c', q) \mid q\,\mathsf{b}\,(1, [\,])\\ \mid c'{\Leftarrow}(r, w) \mid r\,\mathsf{b}\,h_0 \mid h_0\,\mathsf{b}\ - \ \mid w\,\mathsf{b}\,h_0)\end{gathered}$$

TBCHAN.WRITE $\longrightarrow$

$$\begin{gathered}(\nu z)(\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu h_0)(\nu h_1)\\ (E[z] \mid z{\Leftarrow}\mathsf{unit}; \mathsf{readBChan}\ c \mid c{\Leftarrow}(c', q) \mid q\,\mathsf{b}\,(0, [\,])\\ \mid c'{\Leftarrow}(r, w) \mid r\,\mathsf{b}\,h_0 \mid r\,\mathsf{b}\,\mathsf{Item}(\mathsf{True}, h_1) \mid h_1\,\mathsf{b}\ - \ \mid w\,\mathsf{b}\,h_1)\end{gathered}$$

$\longrightarrow$

$$\begin{gathered}(\nu z)(\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu h_0)(\nu h_1)\\ (E[z] \mid z{\Leftarrow}\mathsf{readBChan}\ c \mid c{\Leftarrow}(c', q) \mid q\,\mathsf{b}\,(0, [\,])\\ \mid c'{\Leftarrow}(r, w) \mid r\,\mathsf{b}\,h_0 \mid r\,\mathsf{b}\,\mathsf{Item}(\mathsf{True}, h_1) \mid h_1\,\mathsf{b}\ - \ \mid w\,\mathsf{b}\,h_1)\end{gathered}$$

TBCHAN.READ $\longrightarrow$

$$\begin{gathered}(\nu z)(\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu h_0)\\ (E[z] \mid z{\Leftarrow}\mathsf{True} \mid c{\Leftarrow}(c', q) \mid q\,\mathsf{b}\,(1, [\,])\\ \mid c'{\Leftarrow}(r, w) \mid r\,\mathsf{b}\,h_0 \mid h_0\,\mathsf{b}\ - \ \mid w\,\mathsf{b}\,h_0)\end{gathered}$$

$\text{FUT.DEREF}(\mathsf{ev}) \longrightarrow$

$$(\nu z)(\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu h_0)$$
$$(E[\mathsf{True}] \mid z{\Leftarrow}\mathsf{True} \mid c{\Leftarrow}(c', q) \mid q \,\mathsf{b}\, (1, [\,]) \mid c'{\Leftarrow}(r, w) \mid r \,\mathsf{b}\, h_0 \mid h_0 \,\mathsf{b}\, - \mid w \,\mathsf{b}\, h_0)$$

$\text{GC} \longrightarrow$

$$(\nu c)(\nu q)(\nu c')(\nu r)(\nu w)(\nu h_0)$$
$$(E[\mathsf{True}] \mid c{\Leftarrow}(c', q) \mid q \,\mathsf{b}\, (1, [\,]) \mid c'{\Leftarrow}(r, w) \mid r \,\mathsf{b}\, h_0 \mid h_0 \,\mathsf{b}\, - \mid w \,\mathsf{b}\, h_0)$$

Case 3.6.3.2. In 3.6.2.2 we have already seen that
$\mathbf{thread}(\lambda\_.\mathsf{let}\ q = \mathsf{newQSem}\ n\ \ \mathsf{in}\ \ \mathsf{down}\ q)$ reduces to $(\nu x)(E[\mathsf{True}] \mid x \,\mathsf{b}\, (n - 1, [\,]))$

Proposition 3.6.4.

$$E[\mathbf{thread}(\lambda\_.\mathit{let}\ x = (\mathit{newChan\ unit})\ \mathit{in}\ \ \mathit{writeChan}(x, v);\ \ \mathit{readChan}\ x]$$
$$\sim E[v]$$

Proof. We start with:

$$\mathbf{thread}(\lambda\_.\mathsf{let}\ x = (\mathsf{newChan\ unit})\ \mathsf{in}\ \ \mathsf{writeChan}(x, v);\ \ \mathsf{readChan}\ x)$$

Applying $\text{THREAD.NEW} \longrightarrow$

$$(\nu z)(E[z] \mid z{\Leftarrow}(\lambda\_.\mathsf{let}\ x = (\mathsf{newChan\ unit})\ \mathsf{in}\ \ \mathsf{writeChan}(x, v);\ \ \mathsf{readChan}\ x)z)$$

$\xrightarrow{\beta\text{-CBV}(\mathsf{ev})}$

$$(\nu z)(E[z] \mid z{\Leftarrow}\mathsf{let}\ x = (\mathsf{newChan\ unit})\ \mathsf{in}\ \ \mathsf{writeChan}(x, v);\ \ \mathsf{readChan}\ x)$$

Removing syntactical $\mathsf{let} \ldots\ \mathsf{in}$:

$$(\nu z)(E[z] \mid z{\Leftarrow}(\lambda x.(\mathsf{writeChan}(x, v); \mathsf{readChan}\ x))(\mathsf{newChan\ unit}))$$

$\xrightarrow{\text{TCHAN.NEW}}$

$$(\nu z)(\nu c)(\nu r)(\nu w)(\nu h_1)(E[z] \mid z{\Leftarrow}(\lambda x.(\mathsf{writeChan}(x, v); \mathsf{readChan}\ x))c$$
$$\mid c \Leftarrow (r, w) \mid r \,\mathsf{b}\, h_1 \mid h_1 \,\mathsf{b}\, - \mid w \,\mathsf{b}\, h_1))$$

$\xrightarrow{\beta\text{-CBV}(\mathsf{ev})}$

$$(\nu z)(\nu c)(\nu r)(\nu w)(\nu h_1)(E[z] \mid z {\Leftarrow} \mathsf{writeChan}(c, v); \mathsf{readChan}\ c \\ \mid c \Leftarrow (r, w) \mid r\, \mathsf{b}\, h_1 \mid h_1\, \mathsf{b}\, - \mid w\, \mathsf{b}\, h_1)$$

$\xrightarrow{\text{TCHAN.WRITE}}$

$$(\nu z)(\nu c)(\nu r)(\nu w)(\nu h_1)(\nu h_2)(E[z] \mid z {\Leftarrow} \mathsf{unit}; \mathsf{readChan}\ c \\ \mid c \Leftarrow (r, w) \mid r\, \mathsf{b}\, h_1 \mid h_1\, \mathsf{b}\, \mathsf{Item}(v_1, h_2) \mid h_2\, \mathsf{b}\, - \mid w\, \mathsf{b}\, h_2)$$

$\xrightarrow{\text{GC}}$

$$(\nu z)(\nu c)(\nu r)(\nu w)(\nu h_1)(\nu h_2)(E[z] \mid z {\Leftarrow} \mathsf{readChan}\ c \\ \mid c \Leftarrow (r, w) \mid r\, \mathsf{b}\, h_1 \mid h_1\, \mathsf{b}\, \mathsf{Item}(v_1, h_2) \mid h_2\, \mathsf{b}\, - \mid w\, \mathsf{b}\, h_2)$$

Applying TCHAN.READ:

$$(\nu z)(\nu c)(\nu r)(\nu w)(\nu h_2)(E[z] \mid z {\Leftarrow} v_1 \\ \mid c \Leftarrow (r, w) \mid r\, \mathsf{b}\, h_2 \mid h_2\, \mathsf{b}\, - \mid w\, \mathsf{b}\, h_2)$$

Dereferencing the value of thread z by FUT.DEREF:

$$(\nu z)(\nu c)(\nu r)(\nu w)(\nu h_2)(E[v_1] \mid z {\Leftarrow} v_1 \\ \mid c \Leftarrow (r, w) \mid r\, \mathsf{b}\, h_2 \mid h_2\, \mathsf{b}\, - \mid w\, \mathsf{b}\, h_2)$$

Garbage collection removes thread z $\xrightarrow{\text{GC}}$

$$(\nu c)(\nu r)(\nu w)(\nu h_2)(E[v_1] \mid c \Leftarrow (r, w) \mid r\, \mathsf{b}\, h_2 \mid \ \mid h_2\, \mathsf{b}\, - \mid w\, \mathsf{b}\, h_2)$$

□

Conjecture 3.6.1. Assume there is a program p,

$$\begin{aligned} p := \textbf{thread}^{\tau}(\lambda\_.&\mathsf{let}\ q = \mathsf{newQSem}\ 2 \\ &\mathsf{in}\ \ \textbf{thread}^{\tau}(\mathsf{down}\ q; e; \mathsf{up}\ q) \\ &\quad \textbf{thread}^{\tau}(\mathsf{down}\ q; e; \mathsf{up}\ q) \\ &\quad \textbf{thread}^{\tau}(\mathsf{down}\ q; e; \mathsf{up}\ q)) \end{aligned}$$

THREAD.NEW *and* β-CBV$(\mathsf{ev}) \rightarrow$

$$\begin{aligned} (\nu z)(z {\Leftarrow} &\mathsf{let}\ q = \mathsf{newQSem}\ 2 \\ &\mathsf{in}\ \ \textbf{thread}^{\tau}(\mathsf{down}\ q; e; \mathsf{up}\ q)\ (1) \\ &\quad \textbf{thread}^{\tau}(\mathsf{down}\ q; e; \mathsf{up}\ q)\ (2) \\ &\quad \textbf{thread}^{\tau}(\mathsf{down}\ q; e; \mathsf{up}\ q)\ (3)) \end{aligned}$$

then z is must-convergent if all its child threads are must-convergent.

Proof sketch.

1 z is must-convergent if (1), (2) and (3) are all must convergent.

2 (i) is must-congvergent if e is must-convergent.

3 If $e\Downarrow$ then $e \neq \bot$ and $\widetilde{E}[e] \rightarrow v$, where v is a value.

4 $\forall$ threads (i), $i \leq n$: $(i)\Downarrow$. We must show that $\textbf{thread}^{\tau}(\mathsf{down}\ q; e; \mathsf{up}\ q)\Downarrow$ holds. This follows from the reduction rule of down: (QSEM.DOWN(ev)) converges for a quantity semaphore of count > 0. In our case the semaphore is instantiated with n, so n threads may perform down. Therefore $(\mathsf{down}\ q)\Downarrow$ in these cases. From [2] it follows that e reduces to a value v. v is must-convergent. The specification of quantity semaphores provides that for every successful down-operation there may be one up executed successfully. Therefore $(\mathsf{up}\ q)\Downarrow$ in the case of $i \leq n$.

5 $\forall$ threads (k), $k > n$: $(k)\Downarrow$. From [4] follows, that the first i threads will converge. Then with [4] the next i threads may operate on the semaphore, and so on. Thus, all threads k are must-convergent.

□

Conjecture 3.6.2. Conjecture 3.6.1 holds for n threads.

Conjecture 3.6.3. Let e_1 and e_2 be critical regions, where only one thread may execute in. Then these programs are equivalent:

(1) $p_1 :=$ **thread**$(\lambda_.$let $b =$ newBar 2 in
thread$(\lambda\_.e_1;$ syncBar $b; e_2)$
thread$(\lambda\_.e_2;$ syncBar $b; e_1))$

(2) $p_2 :=$ **thread**$(\lambda_.$let $b_1 =$ **newBuf** True
$b_2 =$ **newBuf** True in
thread$(\lambda\_.$**get** $b_1; e_1;$ **put** $(b_1,$ True$);$ **get** $b_2; e_2;$ **put** $(b_2,$ True$))$
thread$(\lambda\_.$**get** $b_2; e_2;$ **put** $(b_2,$ True$);$ **get** $b_1; e_1;$ **put** $(b_1,$ True$)))$

(3) $p_3 :=$ **thread**$(\lambda_.$let $q_1 =$ newQSem 1
$q_2 =$ newQSem 1 in
thread$(\lambda_.$down $q_1; e_1;$ up $q_1;$ down $q_2; e_2;$ up $q_2)$
thread$(\lambda_.$down $q_2; e_2;$ up $q_2;$ down $q_1; e_1;$ up $q_1))$

(4) $p_4 :=$ **thread**$(\lambda_.$let $c_1 =$ newBChan 1
$c_2 =$ newBChan 1 in
thread$(\lambda_.$writeBChan $(c_1,$ False$); e_1;$ readBChan $c_1;$ writeBChan $(c_2,$ False$); e_2;$ readBChan $c_2)$
thread$(\lambda_.$writeBChan $(c_2,$ False$); e_2;$ readBChan $c_2;$ writeBChan $(c_1,$ False$); e_1;$ readBChan $c_1))$

$$p_1 \sim\ p_2 \sim\ p_3 \sim\ p_4$$

Proof sketch. In every program p_i thread i starts with computing e_i. After finishing the computation both threads take turns on the computation. In the case of buffers, semaphores and bounded channels, each critical region e_i is protected by its own abstraction component. The access of this abstraction is limited to one accessor. By this we ensure that only one thread is allowed to compute e_i. In the case of barrier e_1 and e_2 do not need to be protected with own abstractions. Here one barrier is enough to ensure that threads wait for each other until they may take their turn. □

Summary

In this chapter we firstly proposed some properties concerning the encodings from chapter 2, section 2.3. We proved correctness of some new transformation rules in λ^τ(fchb). Finally, in section 3.6 we have seen coherences between the encodings.

4. Implementation

4.1 Concurrent Haskell

Concurrent Haskell was proposed in [PGF96, Pey09]. It is part of the basic libraries of Haskell. Its stability state is set to 'experimental'[1].

Concurrent Haskell extends Haskell by a primitive `forkIO` and by synchronising variables called `MVar`. MVars behave like one-place buffers. The corresponding operation to **newBuf** is `newEmptyMVar` that creates a new empty MVar. The operation `takeMVar` is equivalent to **get** and `putMVar` behaves like **put** on an empty MVar. Concurrent threads can be started with `forkIO :: IO () -> IO ThreadId`. Applied to an IO-action, a concurrent thread is immediately started to compute the action concurrently.

4.2 The Future Library

We present the library **caf** written in `Haskell`[2] where all[3] discussed primitives from section 2.3 are implemented. The library uses the namespace `Control.Concurrent.Futures`.

At first we present the module `Futures` where several kinds of futures are implemented. Given the module `Futures` and our encodings in $\lambda^\tau(\mathsf{fchb})$, the implementation of the abstractions is not complicated as we will see.

[1]see [cha09].

[2]The name 'future' is already in use for some kinds of *promises*. Thus we use the term **C**oncurrency **A**bstractions using **F**utures for our library. See [Kuk09].

[3]Except the test and set primitive, that is not tested yet. It can be found in Control.Concurrent.Futures.Buffer but is not exposed.

4.2.1 Futures In Haskell

The implementation of futures was proposed in [Sab09]. In his first attempt Sabel distinguishes two types of futures: Explicit and implicit ones. The computation of explicit futures (`EFuture`) must be forced explicitly by calling `force`. This is a bit uncomfortable: The programmer must be careful to force the future at the right moment. Implicit futures are implicitly triggered.

```
efuture :: IO a -> IO (EFuture a)
efuture act =
 do  ack <- newEmptyMVar
     forkIO (act >>= putMVar ack)
     return ack

force :: EFuture a -> IO a
force = takeMVar
```

Figure 4.1: Implementation of explicit futures

An explicit future is represented by an MVar (Fig. 4.1). The creation of an explicit future first creates a new empty MVar and starts a concurrent thread that computes the action resulting in an MVar `ack`. The calling thread immediately gets this new MVar. If it needs the value of the future, then the future must be explicitly forced, which reads the MVar. The calling thread must wait, until the computation of the future has finished. To enable implicity we use `unsafeInterleaveIO` which can delay computations in the IO-monad and breaks sequentiality. Important is that `unsafeInterleaveIO` does not block any other threads. In Fig. 4.2 the implementation of implicit futures is shown. The implicit future needs to know the code that should be computed lazily. The MVar `ack` will be used to store the result of the computation. A new thread will be startet that writes the result of the code into `ack`. The computation of the result itself is being delayed via unsafeInterleaveIO. Once the computation starts it kills the thread and returns the result. Along the implementation of futures the future module is equipped with a global manager. This manager takes care of all introduced futures and ensures the evaluation of all (strict) futures before terminating the main thread. With the manager the usage of futures is easy. The programmer only has to add a global wrapper function `withFuturesDo` around the main function (`main = withFuturesDo code`). The global manager leads to the implementation of strict futures.

```
future :: IO a → IO a
future code = do ack ←newEmptyMVar
                 thread ← forkIO (code >>= putMVar ack)
                 unsafeInterleaveIO (do result ← takeMVar ack
                                        killThread thread
                                        return result)
```

Figure 4.2: Implementation of implicit futures

Strict futures encapsulate a future's computation and the registration of that future in the global Manager.

Finally, there is a need of another type of futures: Lazy futures. Their behaviour is, that there is no computation until the value is needed. If the value of a lazy future is needed then lazy futures behave like strict futures.

```
lazyFuture :: IO a → IO a
lazyFuture code = unsafeInterleaveIO (strictFuture code)

bhandle :: (a -> (a -> IO ()) -> t) -> IO t
bhandle x = do
             f' <- newEmptyMVar
             f  <- lazyFuture  (do
                                 v <- takeMVar f'
                                 putMVar f' v
                                 return v
                               )
             h <- strictFuture (return (\z -> (putMVar f' z)))
             return (x f h)

newhandled :: IO (a -> IO (), a)
newhandled = bhandle (\f -> \h -> (h,f))
```

Figure 4.3: Implementation of lazy futures and handles

With both, lazy and strict futures, the implementation of handles is possible (Fig. 4.3). At first in `bhandle`, it creates a new MVar `f'`. Later, `f'` will bind the handle to the value of the future. A lazy future delays fetching the MVar `f'`s value. Afterwards, the computation component for the handle h is a strict future on a λ-expression that puts an applicated argument into the MVar f'. Finally, `bhandle` returns the tuple (x f h). The function `newhandled` is useful for creating a handle without the purpose of delaying specific code x. It returns the handle and the handled future as a tuple (h,f) which is useful for further needs of either h or f.

4.2.2 Control.Concurrent.Futures.Buffer

```
putBuf :: Buffer a -> a -> IO ()
putBuf (putg,getg,stored,handler) val
      = do (h,f) <- newhandled
           old_value <- exchange putg f
           wait old_value
           exchange stored val
           old_handler <- exchange handler h
           old_handler True
```

Figure 4.4: Implementation of `putBuf` using handles and cells.

As specified in chapter 2.3 a channel consists of buffers. The implementation of buffer in a *lambda calculus with futures* was developped in [SSNSS08]. In [SSNSS08] the type of a buffer is denoted as $\mathsf{buf}\ \tau :: \mathsf{ref\ bool} \rightarrow \mathsf{ref\ bool} \rightarrow \mathsf{ref}\ \tau \rightarrow \mathsf{ref}(\mathsf{bool} \rightarrow \mathsf{unit})$. In Haskell, we implement a buffer as of type `type Buffer a = (Cell Bool, Cell Bool, Cell a, Cell (Bool -> IO ()))` (Fig. 4.4 and 4.5). We have our own cell type because we need an atomic exchange operation on cells. To provide this exchange operation we implement a cell as `Cell a :: MVar a` and use `swapMVar` as an exchange on cells. We do not need any other operations from the MVars implementation. A new Buffer is initialised by `True` on the 'putg' cell. The cells 'getg' and 'stored' are assigned to futures. The cell 'handler' is assigned to the handle of 'getg's future. The function `putBuf` creates new handle components for the next put-operation. The `old_value` is True, if a put is allowed at this time. Therefore, the function waits on this value, until it becomes true. Not till then, there is an exchange of the cell 'stored' with the new value. The new handle is exchanged with the old one that will bind to true. The function `getBuf` creates two new handle components: One for the storage cell and the other for the next put operation. It exchanges 'getg' with the new future and waits on the `old_value` - that is a future - to become true. If it becomes true, an exchange of the storage cell with the new future is executed. The new handle is exchanged with the old one that will be bounnd to true. Finally it returns the read value. Note, that if we want to use a buffer, we need to use `withFuturesDo` in our main function.

```
getBuf :: Buffer a -> IO a
getBuf (putg,getg,stored,handler)
     = do (h,f) <- newhandled
          (h',f') <- newhandled
          old_value <- exchange getg f
          wait old_value
          val <- exchange stored f'
          old_handler <- exchange handler h
          old_handler True
          return val
```

Figure 4.5: Implementation of `getBuf` in Haskell.

4.2.3 Control.Concurrent.Futures.Chan

```
newChan :: IO (Chan a)
newChan = do hole <- newBuf
             read_end <- newBuf
             write_end <- newBuf
             putBuf read_end hole
             putBuf write_end hole
             return (Chan read_end write_end)
```

Figure 4.6: Implementation of `newChan` in `Haskell`.

As specified in chapter 2.3 a channel is of type $(\mathsf{buf}\ (\mathsf{buf}\ (\mathsf{item}\ \tau)), \mathsf{buf}(\mathsf{buf}\ (\mathsf{item}\ \tau)))$. We directly translate this encoding into Haskell as: `type ChanType a = ((Buffer (ItemType a)), (Buffer (ItemType a)))`. We also need a type 'item' `type ItemType a = (Buffer (Item a))` and a corresponding data type 'Item' `data Item a = Item a (ItemType a)`. Given the module `Futures` and the `Buffer`, the implementation of channels is straightforward. Fig. 4.6 and Fig. 4.7 show the code. The function `newChan` creates three new empty buffers and returns the tuple of the write- and read-buffer. `writeChan` creates a new buffer being the new write-end. Then it takes the value of the old write-end, puts the new buffer into it and writes a new item containing the value `val` into it. The function `readChan` reads the item out of the read-end. Then it writes the item's content back to the read-end and returns the value `val`. Note, that if we want to use these channels, we need to use `withFuturesDo` in our main function.

```
writeChan :: Chan a -> a -> IO ()
writeChan (read_end,write_end) val
         = do new_hole <- newBuf
              old_hole <- getBuf write_end
              putBuf write_end new_hole
              putBuf old_hole (Item val new_hole)
readChan :: Chan a -> IO a
readChan (read_end,write_end)
         = do chan_head <- getBuf read_end
              (Item val content) <- getBuf chan_head
              putBuf read_end content
              return val
```

Figure 4.7: Implementation of `readChan` and `writeChan` in `Haskell`.

4.2.4 Implementing Quantity Semaphores In Haskell

The module `Control.Concurrent.Futures.QSem` as well as `Control.Concurrent.Futures.HQSem` contain an implementation of quantity semaphores. The first one uses a waiting queue of buffers. The implementation of the functions `up` and `down` is shown in Fig. 4.8. In the case of quantity semaphores with a waiting queue on handles we have the type `type HQSem = Buffer (Int, [Bool -> IO ()])`. Due to the new type it is `newQSem :: Int -> IO (HQSem)`. There is no change in the implementation of newQSem, but of `up` and `down` as we see in Fig. 4.9. Compared with Haskell, our quantity semaphores `QSem` and `HQSem` are convenient to `QSemN`. Haskell's `QSem` implementation suits our implementation of `Buffer`.

```
up :: QSem -> IO ()
up qsem = do
 (cnt,ls) <- getBuf qsem
 case ls of
      [] -> do putBuf qsem (cnt+1,ls)
    x:xs -> do putBuf x True
               putBuf qsem (cnt,ls)
```

```
down :: QSem -> IO Bool
down qsem = do
 b <-getBuf qsem
 case b of
     (cnt,ls) -> case (cnt==0) of
         True -> do b1 <- newBuf
                    putBuf qsem (cnt,b1:ls)
                    getBuf b1
         False -> do putBuf qsem (cnt-1,ls)
                     return True
```

Figure 4.8: Implementation of quantity semaphores using buffers.

```
up :: HQSem -> IO ()
up qsem = do
 b <- getBuf qsem
 case b of
  (cnt,ls) -> case ls of
     [] -> do putBuf qsem (cnt+1,ls)
   x:xs -> do x True
              putBuf qsem (cnt,ls)
```

```
down :: HQSem -> IO (Bool)
down qsem = do
 b <-getBuf qsem
 case b of
  (cnt,ls) -> case (cnt==0) of
     True -> do (h,f) <- newhandled
        putBuf qsem (cnt,h:ls)
        (wait f)
    False -> do putBuf qsem (cnt-1,ls)
        return True
```

Figure 4.9: Implementation of quantity semaphores using handles.

4.2.5 Control.Concurrent.Futures.BChan

The bounded channel module are now briefly discussed. With the new modules `QSem` and `Chan` we can implement bounded channels as proposed in the encoding section 2.3. The type of the bounded channel is `type BChan a = (Chan a, QSem)`. The implementation in (Fig. 4.10 and Fig. 4.11) goes straightforward using quantity semaphores and channels. We used the QSem and not HQSem. Note that further research could profile the runtime properties of the two versions of semaphores. After that we could choose the more efficient one for the implementation of bounded channels. Note that here as well, we need to use `withFuturesDo` in our main function.

```
readBChan :: BChan a -> IO a
readBChan (chan, sem)
         = do
           up sem
           readChan chan
```

```
writeBChan :: BChan a -> a -> IO ()
writeBChan (chan, sem) val
          = do down sem
               writeChan chan val
```

Figure 4.10: Implementation of operations on a bounded channel in Haskell.

```
newBChan :: Int -> IO (BChan a)
newBChan n
       = do chan <- newChan
            qsem <- newQSem n
            return (chan , qsem)
```

Figure 4.11: Implementation of `newBChan` in Haskell.

4.2.6 Implementing Barrier In Haskell

The module `Control.Concurrent.Futures.Barrier` provides a type for a barrier `type Bar a = (Buffer a, Buffer [Bool -> IO ()], Int)` and the functions `newBar` and `syncBar`. In the case of `syncBar` we made an important change compared to the implementation in $\lambda^\tau(\mathsf{fchb})$: After the evaluation of `(act==1)` to True, it will firstly run `openBar` before putting the new values into the buffers `cnt_act` and `cnt_fin`. Note that we need to use `withFuturesDo` in our main function when using a barrier.

```
syncBar :: (Buffer Int, Buffer [Bool -> IO ()], Int) -> IO Bool
syncBar (cnt_act,cnt_fin,k) = do
 act <- Buffer.getBuf cnt_act
 fin <- Buffer.getBuf cnt_fin
 case (act==1) of
  True -> do
          openBar fin
          Buffer.putBuf cnt_act (act-1)
          Buffer.putBuf cnt_fin []
          return True
  False -> do
           Buffer.putBuf cnt_act (act-1)
           (h,f) <- Futures.newhandled
           Buffer.putBuf cnt_fin (h:fin)
           wait f
```

Figure 4.12: Implementation of `syncBar` in Haskell.

4.3 Using The Library

We install our future library by running the command runhaskell Setup install[4] from the folder where we unpacked the tarball of the library. This command registers the library as a new package for ghc[5]. After installing the package we can simply import the new modules from the distribution package in our new `Haskell` files. To see which modules are provided by our package we use the command ghc-pkg field caf exposed-modules. For example writing a new FutureApp that uses concurrency abstractions from the future package, we just import the abstractions like this:

```
module FutureApp where
 import Control.Concurrent.Futures.Buffer
 import Control.Concurrent
```

Running The Examples. We added a module `Examples` where one can test the abstractions. For each abstraction there is one example just using `do` and one using `withFuturesDo`. In the case of using `do` the main thread terminates after a while. If we use `withFuturesDo` as recommended, the main thread never stops before its child-threads have ended their computation. See the documentation for further information.

Uninstalling The Distribution Package. The command ghc-pkg list shows us a list of all installed packages. To unregister the package **caf** we use ghc-pkg unregister caf-1.0. caf-1.0 is the name of the future library where the postfix '-1.0' indicates its version.

Summary

In this chapter we have seen the implementation of concurrency abstraction from $\lambda^\tau(\mathsf{fchb})$ in `Haskell`. Owing to the future module from [Sab09] and the encoding of the abstractions in $\lambda^\tau(\mathsf{fchb})$ this was straightforward.

[4]The command runhaskell might be runhugs, runghc or runnhc.
[5]The Glasgow Haskell Compiler.

Conclusion

We have extended *the lambda calculus with futures* by five new concurrency primitives. We proposed a translation for each primitive to $\lambda^{\tau}(\mathsf{fchb})$. Resulting transformation rules for new concurrency abstractions have been proven as correct. This result illustrates one of the advantages of good observational semantics. We demonstrated equivalence on concurrent programs. Finally, the Haskell implementation can be seen as a proof of concept. We wrote it really quickly and straightforward due to the good encoding.

The encoding of the barrier is very simple. Barrier could be even more powerful. Compared to Haskell's barrier implementation our implementation lacks a count of the current phase. It also lacks the possibility to join the barrier. In conclusion, we will look forward to improve the encodings.

In the proof section, further research must verify the correctness of TOPEN to be evidence of the correctness of the barrier transformation rule TSYNC. Regarding the quantity semaphores we only considered the handle-version. We presume that there are much more equivalences between the resprective concurrency abstractions. Finding equity helps to challenge the research on showing properties of the proposed translations of $\lambda^{\tau}(\mathsf{fchb}) + x \rightarrow \lambda^{\tau}(\mathsf{fchb})$.

The `Future` package should be committed to the haskell community with intent to share it. Approaching a stable state it has the potential of being integrated into Haskell.

In conclusion, *the lambda calculus with futures*, which was designed to be a formal system for proving equivalence of concurrent programs, turns out to be as powerful as expected.

List of Figures

Bibliography

[Bac78] John Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, August 1978.

[cha09] Haskell - haskellwiki, February 2009. `http://www.haskell.org`.

[INR09] INRIA. Jocaml, February 2009. `http://jocaml.inria.fr`.

[Kuk09] Christopher Edward Kuklewicz. Hackagedb - future-2.0.0, March 2009. `http://hackage.haskell.org/cgi-bin/hackage-scripts/package/future`.

[NSS06] Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A concurrent lambda calculus with futures. *Theoret. Comput. Sci.*, 364(3):338–356, November 2006.

[Pey09] S. Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. 2009.

[PGF96] Simon L. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *23rd ACM Symposium on Principles of Programming Languages*, pages 295–308, St Petersburg Beach, Florida, January 1996. ACM.

[Rep99] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, New York, NY, USA, 1999.

[Res09a] Microsoft Research. C omega, February 2009. `http://research.microsoft.com/en-us/um/cambridge/projects/comega/`.

[Res09b] Microsoft Research. F sharp, February 2009. `http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/`.

[Sab09] David Sabel. Implementing concurrent futures in concurrent haskell. working draft, available at http://www.ki.informatik.uni-frankfurt.de/persons/sabel/futures.pdf, January 2009.

[SSNSS08] Jan Schwinghammer, David Sabel, Joachim Niehren, and Manfred Schmidt-Schauß. On proving the equivalence of concurrency primitives. Frank report 34, Institut für Informatik. Fachbereich Informatik und Mathematik. J. W. Goethe-Universität Frankfurt am Main, October 2008.

[SSNSS09] Jan Schwinghammer, David Sabel, Joachim Niehren, and Manfred Schmidt-Schauß. On correctness of buffer implementations in a concurrent lambda calculus with futures. Frank report 37, Institut für Informatik. Fachbereich Informatik und Mathematik. Goethe-Universität Frankfurt am Main, March 2009.

[Tan01] Andrew S. Tanenbaum. *Modern Operating Systems (2nd Edition) (GOAL Series)*. Prentice Hall, 2 edition, March 2001.