On Proving the Equivalence of Concurrency Primitives

Jan Schwinghammer[1], David Sabel[2], Joachim Niehren[3], and Manfred Schmidt-Schauß[2]

[1] Saarland University, Saarbrücken, Germany
[2] Goethe-Universität, Frankfurt, Germany
[3] INRIA, Lille, France, Mostrare Project

Technical Report Frank-34

Research group for Artificial Intelligence and Software Technology
Institut für Informatik,
Fachbereich Informatik und Mathematik,
Johann Wolfgang Goethe-Universität,
Postfach 11 19 32, D-60054 Frankfurt, Germany

13. October 2008

Abstract. Various concurrency primitives have been added to sequential programming languages, in order to turn them concurrent. Prominent examples are concurrent buffers for Haskell, channels in Concurrent ML, joins in JoCaml, and handled futures in Alice ML. Even though one might conjecture that all these primitives provide the same expressiveness, proving this equivalence is an open challenge in the area of program semantics. In this paper, we establish a first instance of this conjecture. We show that concurrent buffers can be encoded in the lambda calculus with futures underlying Alice ML. Our correctness proof results from a systematic method, based on observational semantics with respect to may and must convergence.

1 Introduction

Modern concurrent programming languages extend sequential languages with concurrent threads, and concurrency primitives for controlling their interactions. Computation in each thread is sequential. Examples for concurrency primitives are concurrent buffers in Haskell [4], channels in Concurrent ML [11], handled futures in Alice ML [14], and joins in JoCaml [3].

Even though one might conjecture that all these primitives provide the same expressiveness, proving such equivalences is an open challenge in the area of programming language semantics. Encodings are usually not difficult to find for a fixed common sequential base language. In fact, synchronization mechanisms are

often provided through libraries of the programming languages, and implemented in terms of lower-level primitives. The difficult part is to introduce appropriate notions of program semantics, and corresponding proof techniques for showing correctness of an encoding. Recent progress allows us to settle these questions, as we will illustrate in the present paper.

Our starting point is to view the implementation of synchronization constructs by low-level primitives as a *translation* $T : \mathcal{C} \to \mathcal{C}'$ between two languages. Correctness of an implementation then becomes a question about relating programs and their images under this translation. We consider the adequacy of a translation as the appropriate correctness condition [12, 13, 17]. It can be defined for any kind of program semantics that gives rise to an equivalence relation $=_{\mathcal{C}}$ on the programs of $\mathcal{C}$ (e.g. a denotational, bisimulation-based, or operationally-defined observational semantics): a translation is adequate if all (equally typed) programs with equivalent translations are equivalent, i.e., $T(p_1) =_{\mathcal{C}'} T(p_2)$ implies $p_1 =_{\mathcal{C}} p_2$. If additionally the converse holds, then T is called fully abstract.

We assume observational semantics based on operationally-defined forms of may- and must-convergence [2, 7, 1]. Their combination properly captures the non-determinism arising in concurrent programming languages [5, 15].

In this paper, we consider the lambda calculus with futures, reference cells and handles as base language [6], which was introduced as foundation of the operational semantics of Alice ML [14]. We define a typed version of the lambda calculus with futures called $\lambda^{\tau}(\mathsf{fc})$, which extends the untyped original version $\lambda(\mathsf{f})$ [6, 5] by recursive data types, algebraic constructor terms and case-expressions. We then consider a particular encoding of concurrent buffers in $\lambda^{\tau}(\mathsf{fc})$. In turn, such buffers can be used to implement more complex concurrency abstractions, like buffered channels (see [10]). We specify the semantics of buffers by extending the syntax and operational semantics; our main contribution is the proof of full abstraction of the encoding (which relies on typing). This result tells us that the implementation is faithful to the specification, with respect to the observable behavior. As a corollary, we can derive equivalences for concurrent buffers from equivalences valid for $\lambda^{\tau}(\mathsf{fc})$.

Our adequacy and full abstraction proof is based on a systematic method comprising two parts. First of all, we have to establish appropriate equivalences for the base language $\lambda^{\tau}(\mathsf{fc})$. For $\lambda(\mathsf{f})$ a rich collection of equivalences were proved valid on the basis of a context lemma and diagram based methods [5]. In order to lift them to $\lambda^{\tau}(\mathsf{fc})$, we encode $\lambda^{\tau}(\mathsf{fc})$ into $\lambda(\mathsf{f})$ via a simplified intermediate calculus, and prove the adequacy of both translations. In the second part, we prove that the encoding of buffers preserves and reflects may- and must-convergence. The proof uses commutation techniques for reduction steps as well as valid equivalences previously shown for $\lambda^{\tau}(\mathsf{fc})$. These are necessary to prove invariants of the implicit queuing mechanisms that arise in the buffer implementation. Using compositionality of the translation this lets us conclude full abstraction [16, 17].

Questions of expressiveness have been addressed mainly in pi-calculus and basic process calculi [9]; we are not aware of previous work on formally relating synchronization primitives in concurrent high-level languages. Similar issues

(concerning properties of translations) arise in the verification of compilers, but usually only (simpler) simulation properties for closed programs, rather than open program fragments, are established. An alternative approach to equational correctness proofs are Hoare-style program logics; recently, concurrent separation logic has been used to prove properties of concurrent data structures [8].

2 The Lambda Calculus with Futures

In this section we introduce $\lambda^\tau(\mathsf{fc})$, a typed variant of the λ-calculus with futures $\lambda(\mathsf{f})$ from [6]. The calculus extends $\lambda(\mathsf{f})$ by recursive data types, algebraic constructor terms and the corresponding case-expressions.

Syntax. We use vector notation for many syntactic categories, so we write $\vec{a}$ for a sequence of objects $a_1, \ldots, a_n$ where $n \geq 0$. Let a signature $\Sigma = (\mathcal{K}, \mathcal{D})$ consist of two finite ranked sets, a set of type constructors $\kappa \in \mathcal{K}$ and a set of data constructors $k \in \mathcal{D}$, with fixed arities $ar(\kappa) \geq 0$ and $ar(k) \geq 0$. A simple type $\tau \in \mathcal{T}$ is a term built from function symbols in $\mathcal{K} \cup \{\mathsf{ref}\ , \rightarrow\}$, as defined in Fig. 1. We assume that each data constructor $k \in \mathcal{D}$ has a unique type of the form $\tau'_1 \rightarrow \ldots \rightarrow \tau'_{ar(k)} \rightarrow \kappa(\vec{\tau})$. In this case, we write $in_{\mathcal{D}}(k) = \tau'_1 \ldots \tau'_{ar(k)}$ for the sequence of input types of k. Let $\mathcal{D}(\kappa(\vec{\tau}))$ be the set of data constructors in $\mathcal{D}$ with output type $\kappa(\vec{\tau})$. We assume that none of these sets is empty, so that we obtain a partition into finite non-empty sets $\mathcal{D} = \bigcup_{\kappa(\vec{\tau}) \in \mathcal{T}} \mathcal{D}(\kappa(\vec{\tau}))$. Moreover, we assume that the signature contains constructors for (n-ary) products: there are n-ary type constructors $(\cdot \times \ldots \times \cdot)$ in $\mathcal{K}$ and data constructors $\langle \cdot, \ldots, \cdot \rangle$ in $\mathcal{D}(\tau_1 \times \ldots \times \tau_n)$ with $in_{\mathcal{D}}(\langle \cdot, \ldots, \cdot \rangle) = \vec{\tau}$, for all types $\vec{\tau} = \tau_1, \ldots, \tau_n$.

The syntax of $\lambda^\tau(\mathsf{fc})$-expressions is defined in Fig. 1, where *Var* is a fixed infinite set of variables. It consists of two layers: a level of λ-expressions $e \in \mathit{Exp}$ (using constructors in $\mathcal{D}$) for sequential computation within threads, and a level of processes $p \in \mathit{Proc}$ that compose threads in parallel and record the state of the system. For typed case-expression $\mathbf{case}_{\kappa(\vec{\tau})}\ e\ \mathbf{of}\ \pi_1 \Rightarrow e_1 \mid \ldots \mid \pi_m \Rightarrow e_m$, we assume that each constructor $k \in \mathcal{D}$ appears at most once in the patterns π_i, and the set of all these constructors is exactly $\mathcal{D}(\kappa(\vec{\tau}))$. Thus the patterns are non-overlapping and exhaustive. Like abstractions, patterns act as binders in a branch $\pi \Rightarrow e$, and all the variables appearing in π must be distinct. The set of free variables of e is denoted by $\mathit{fv}(e)$ (similarly $\mathit{fv}(p)$ for processes p), expressions and processes are identified up to consistent renaming of bound variables, and we write $e[e'/x]$ for the (capture-free) substitution of e' for x in e.

New operations in expressions are introduced by (higher-order) constants: the constants **thread**, **lazy** and **handle** serve for introducing eager threads, lazy threads, and handles, each of them together with a future. The constant **cell** introduces reference cells, and the expression $\mathbf{exch}(e_1, e_2)$ expresses atomic exchange of cell values. Note that we distinguish between constants and data constructors $k \in \mathcal{D}$ – the latter must always be fully applied. Values v are defined as usual in a call-by-value λ-calculus.

As in the π-calculus, processes p are composed from smaller components by parallel composition $p_1 \mid p_2$ and new name creation $(\nu x)p$. The latter is a variable

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

Fig. 1. Types, expressions and processes of $\lambda^{\tau}(\mathsf{fc})$, where $m \geq 0$

$$
\begin{aligned}
&p_1 \mid p_2 \ \equiv\ p_2 \mid p_1 && (p_1 \mid p_2) \mid p_3 \ \equiv\ p_1 \mid (p_2 \mid p_3) \\
&(\nu x)(\nu y)p \equiv (\nu y)(\nu x)p && (\nu x)(p_1) \mid p_2 \equiv (\nu x)(p_1 \mid p_2) \quad \text{if } x \notin \mathit{fv}(p_2)
\end{aligned}
$$

Fig. 2. Structural congruence of processes

binder. A *structural congruence* $\equiv$ on processes is defined by the axioms in Fig. 2. We distinguish five types of components that have no direct correspondence in π-calculus: (eager concurrent) threads $x \Leftarrow e$ will eventually bind future x to the value of expression e unless it diverges or suspends; x is called a *concurrent future*. Lazy threads $x \overset{susp}{\Longleftarrow} e$ are suspended computations that will start once the proper value of x is needed elsewhere; we call x a lazy future. Cells $x\,\mathsf{c}\,v$ associate (a memory location) x to a value v. Handle components $y\,\mathsf{h}\,x$ associate handles y to futures x, so that y can be used to assign a value to x. We call x a future handled by y, or more shortly a *handled future*. Finally, a used handle component $y\,\mathsf{h}\,\bullet$ indicates that y is a handle that has already been used to bind its associated future.

Contexts and Operational Semantics. The operational semantics defines an evaluation strategy via (evaluation) contexts in which reduction rules apply. A *context* is a process or expression with a single occurrence of the hole marker $[\,]$. The hole marker for terms can only occur in positions where the grammar allows arbitrary expressions: for instance, in a cell $x\,\mathsf{c}\,v$ the position of a hole can only be in subterms of v that are abstractions. The result of placing e in context C (possibly capturing free variables of e) is written $C[e]$.

Fig. 3 defines *evaluation contexts* (ECs) E and *future ECs* F as particular contexts. ECs encode the standard call-by-value, left-to-right reduction strategy, while future ECs control dereferencing operations on futures and the triggering of suspended threads. The small-step reduction relation $p \to p'$ is the least binary relation on processes satisfying the rules in Fig. 4.[4] We write $\overset{\mathsf{ev}}{\longrightarrow}$ for $\to$ when we want to distinguish reductions from *transformations*.

[4] Here we use call-by-value β-reduction as in [6]. In [5] we used a sharing variant of call-by-value β-reduction. To distinguish between both calculi we denote the calculus of [6] with $\lambda(\mathsf{f})$ and the calculus of [5] with $\lambda(\mathsf{f}')$. The equational theories of both calculi are the same. In Appendix A we formalize this claim by showing full-abstractness for the identity translations between both calculi.

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

Fig. 3. Evaluation contexts

The rule (THREAD.NEW(ev)) spawns a new eager thread $x \Leftarrow e$ where x may occur in e, so it may be viewed as a recursive declaration $x = e$. Similarly, (LAZY.NEW(ev)) creates a new suspended computation $x \overset{susp}{\Longleftarrow} e$. Dereferencing of future values (FUT.DEREF(ev)) and triggering of suspended computations (LAZY.TRIGGER(ev)) is controlled by future evaluation contexts F. The rule (HANDLE.NEW(ev)) introduces handle components $y\,\mathsf{h}\,x$ with static scope in e; the application $x\,v$ in (HANDLE.BIND(ev)) "consumes" the handle x and binds y to v, resulting in a used handle $x\,\mathsf{h}\,\bullet$ and thread $x \Leftarrow v$. Rule (CELL.NEW(ev)) creates new cells $z\,\mathsf{c}\,v$ with contents v. The exchange operation $\mathbf{exch}(z, v_1)$ writes v_1 to the cell and returns the previous contents. Since this is an atomic operation, no other thread can interfere.

Handled futures are the basic synchronization construct in $\lambda^{\tau}(\mathsf{fc})$: a thread may block on a future x until a second thread uses the associated handle and provides a value for x.

Typing. Types are assigned to both expressions and processes. The typing relation for expressions, $\Gamma \vdash e : \tau$ where Γ is a type context that associates types to distinct identifiers, is defined by the usual typing rules of simply typed lambda calculus together with the types for constructors and the typing rules for **exch** and case expressions given in Fig. 5. The types of each constant c are given by a polymorphic type scheme, with $\mathit{TypeOf}(c)$ denoting the set of all instances.

On the level of processes, types ensure a number of well-formedness conditions. For instance, there is a unique binding for every (concurrent, lazy, or handled) future and each reference cell. Intuitively, the judgement $\Gamma \vdash p : \Gamma'$ expresses that p has free variables as described by Γ which need to be provided externally, while p provides visible bindings described by Γ' which may be used externally. Fig. 6 gives the inference rules defining this judgement. The rule for parallel compositions can be understood by noting the similarity to (mutually recursive) declarations. It requires that the variables introduced by p_1 and p_2 are disjoint, by the convention that Γ, Γ' is defined only if the domains of Γ and Γ' are disjoint. The binding operator $(\nu x)p$ restricts the observational scope of an introduced variable x to p; the binding is therefore removed from Γ'.

For a more detailed explanation of the type system we refer to [6] and note only that the type safety theorem extends to $\lambda^{\tau}(\mathsf{fc})$.

Reduction rules.

(β-CBV(ev))	$E[(\lambda y.e)\, v] \to E[e[v/y]]$
(THREAD.NEW(ev))	$E[\mathbf{thread}\, v] \to (\nu z)(E[z] \mid z \Leftarrow v\, z)$
(FUT.DEREF(ev))	$F[x] \mid x \Leftarrow v \to F[v] \mid x \Leftarrow v$
(HANDLE.NEW(ev))	$E[\mathbf{handle}\, v] \to (\nu z)(\nu z')(E[v\, z\, z'] \mid z'\, \mathsf{h}\, z)$
(HANDLE.BIND(ev))	$E[x\, v] \mid x\, \mathsf{h}\, y \to E[\mathbf{unit}] \mid y \Leftarrow v \mid x\, \mathsf{h}\, \bullet$
(CELL.NEW(ev))	$E[\mathbf{cell}\, v] \to (\nu z)(E[z] \mid z\, \mathsf{c}\, v)$
(CELL.EXCH(ev))	$E[\mathbf{exch}(z, v_1)] \mid z\, \mathsf{c}\, v_2 \to E[v_2] \mid z\, \mathsf{c}\, v_1$
(LAZY.NEW(ev))	$E[\mathbf{lazy}\, v] \to (\nu z)(E[z] \mid z \overset{susp}{\Longleftarrow} v\, z)$
(LAZY.TRIGGER(ev))	$F[x] \mid x \overset{susp}{\Longleftarrow} e \to F[x] \mid x \Leftarrow e$
(CASE.BETA(ev))	$E[\mathbf{case}\ k_j(v_1, \ldots, v_{ar(k_j)})\ \mathbf{of}\ (k_i(x_1, \ldots, x_{ar(k_i)}) \Rightarrow e_i)^{i=1\ldots n}] \to E[e_j[v_1/x_1, \ldots, v_{ar(k_j)}/x_{ar(k_j)}]]$

Distinct variable convention. We assume that all processes to which rules apply satisfy the distinct variable convention, and that all new binders use fresh variables (z and z'). Reduction results will satisfy the distinct variable convention, if after β-CBV(ev), CASE.BETA(ev) and FUT.DEREF(ev) where values with bound variables may be copied, α-renaming is performed before applying the next rule.

Closure. Rule application is closed under structural congruence and process ECs D: if $p_1 \equiv D[p_1']$, $p_1' \to p_2'$, and $D[p_2'] \equiv p_2$ then $p_1 \to p_2$.

Fig. 4. One-step reduction relation of $\lambda^{\tau}(\mathsf{fc})$, denoted by $\to$ or $\overset{\mathsf{ev}}{\longrightarrow}$

Observations and Contextual Equivalence. A process p is *successful* (meaning it has terminated successfully) if in every component $x \Leftarrow e$ of p, the identifier x is bound (possibly via a chain $x \Leftarrow x_1 \mid x_1 \Leftarrow x_2 \mid \ldots \mid x_{n-1} \Leftarrow x_n$) to a non-variable value, a cell, a lazy future, a handle, or a handled future. Hence, in a non-failing computation, every future eventually refers to a "proper" value. For instance, $x \Leftarrow \lambda y.y$, $x \Leftarrow y \mid y \Leftarrow \langle x, x\rangle$ and $x \Leftarrow y \mid y\, \mathsf{c}\, z$ are successful, while $x \Leftarrow x$ (a black hole, in the terminology of call-by-need calculi) and $x \Leftarrow yx \mid y \Leftarrow xy$ (a deadlocked process) are ruled out.

We use $p{\downarrow}$ to indicate that p is *may-convergent*, i.e., that there is a sequence of reductions $p \to^* p'$ such that p' is successful, and $p{\Downarrow}$ if the process is *must-convergent*, meaning that all reduction descendants p' of p are may-convergent. Dually, we call p *must-divergent* ($p{\Uparrow}$) if it has no reduction descendant that succeeds, and *may-divergent* ($p{\uparrow}$) if some reduction descendant of p is must-divergent. Thus, $p{\Uparrow} \Leftrightarrow \neg p{\downarrow}$ and $p{\uparrow} \Leftrightarrow \neg p{\Downarrow}$.

The typing of terms and processes gives rise to a notion of typed contexts: for $\Delta = (\Gamma, \Gamma')$ a pair of type environments we define the judgement $\Delta \vdash p$ by $\Gamma \vdash p : \Gamma'$, and let $\mathit{Ctxt}(\Delta)$ be the set of contexts C for which $\Delta \vdash p$ implies the existence of Δ' with $\Delta' \vdash C[p]$. Now, for $obs \in \{{\downarrow}, {\Downarrow}\}$, we define contextual approximation relations between processes p_1 and p_2 with $\Delta \vdash p_i$ by:

$$\Delta \vdash p_1 \leq_{obs} p_2 \quad \Leftrightarrow \quad \forall D \in \mathit{Ctxt}(\Delta).\ D[p_1] \in obs \Rightarrow D[p_2] \in obs$$

$$\textbf{unit} : \mathsf{unit} \quad \textbf{thread} : (\alpha{\to}\alpha){\to}\alpha \quad \textbf{lazy} : (\alpha{\to}\alpha){\to}\alpha$$
$$\textbf{handle} : (\alpha{\to}(\alpha{\to}\mathsf{unit}){\to}\beta){\to}\beta \quad \textbf{cell} : \alpha{\to}\mathsf{ref}\ \alpha$$

$$\frac{\tau \in \mathit{TypeOf}(c)}{\Gamma \vdash c : \tau} \qquad \frac{\Gamma \vdash e_1 : \mathsf{ref}\ \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \textbf{exch}(e_1, e_2) : \tau} \qquad \frac{k \in \mathcal{D}(\kappa(\vec{\tau})) \qquad \Gamma \vdash \vec{e} : in_{\mathcal{D}}(k)}{\Gamma \vdash k(\vec{e}) : \kappa(\vec{\tau})}$$

$$\frac{\Gamma \vdash e : \kappa(\vec{\tau}) \qquad \mathcal{D}(\kappa(\vec{\tau})) = \{k_1, \ldots, k_n\} \qquad \forall i = 1 \ldots n.\ \ \Gamma, \vec{x_i} : in_{\mathcal{D}}(k_i) \vdash e_i : \tau}{\Gamma \vdash (\textbf{case}_{\kappa(\vec{\tau})}\ e\ \textbf{of}\ (k_i\, \vec{x_i} \Rightarrow e_i)^{i=1\ldots n}) : \tau}$$

Fig. 5. Some typing rules for expressions

$$\frac{\Gamma, \Gamma_1 \vdash p_1 : \Gamma_2 \qquad \Gamma, \Gamma_2 \vdash p_2 : \Gamma_1}{\Gamma \vdash p_1 \mid p_2 : \Gamma_1, \Gamma_2} \qquad \frac{\Gamma, x{:}\tau \vdash e : \tau}{\Gamma \vdash x \overset{(susp)}{\Longleftarrow} e : (x{:}\tau)} \qquad \frac{\Gamma, x{:}\mathsf{ref}\ \tau \vdash v : \tau}{\Gamma \vdash x\, \mathsf{c}\, v : (x{:}\mathsf{ref}\ \tau)}$$

$$\frac{\Gamma \vdash p : \Gamma'}{\Gamma \vdash (\nu x)p : \Gamma' - x} \qquad \frac{y \notin \mathit{dom}(\Gamma)}{\Gamma \vdash y\, \mathsf{h}\, \bullet : (y{:}\tau \to \mathsf{unit})} \qquad \frac{x, y \notin \mathit{dom}(\Gamma)}{\Gamma \vdash y\, \mathsf{h}\, x : (x{:}\tau, y{:}\tau \to \mathsf{unit})}$$

Fig. 6. Typing of processes

We write $\Delta \vdash p_1 \leq p_2$ if both $\Delta \vdash p_1 \leq_{\downarrow} p_2$ and $\Delta \vdash p_1 \leq_{\Downarrow} p_2$ hold, and $\Delta \vdash p_1 \sim p_2$ if both $\Delta \vdash p_1 \leq p_2$ and $\Delta \vdash p_2 \leq p_1$ hold.

Translations. An abstract calculus with observational semantics in the framework from [17] consists of sets of processes, contexts, types and convergence predicates. The calculus $\lambda^{\tau}(\mathsf{fc})$ is a calculus in this general framework, where Δ's serve as types. The (untyped) calculus $\lambda(\mathsf{f})$ is another such calculus with a single type that is universal.

A *translation T between two such calculi* maps types to types, processes to processes, and contexts to contexts, such that their types correspond. A translation T between calculi $\mathcal{C}$ and $\mathcal{C}'$ is *adequate* if T reflects operational approximation, i.e. iff $T(\Delta) \vdash T(p_1) \leq_{\mathcal{C}'} T(p_2) \ \Rightarrow\ \Delta \vdash p_1 \leq_{\mathcal{C}} p_2$ for all Δ and p_1, p_2 such that $\Delta \vdash p_i$. If T additionally preserves inequations, i.e. iff for all Δ, p_1, p_2 with $\Delta \vdash p_1, \Delta \vdash p_2$ the equivalence $\Delta \vdash p_1 \leq_{\mathcal{C}} p_2 \ \Leftrightarrow\ T(\Delta) \vdash T(p_1) \leq_{\mathcal{C}'} T(p_2)$ holds, then it is *fully abstract*. A translation is *convergence equivalent* iff $T(p)\!\downarrow \Leftrightarrow p\!\downarrow$ and $T(p)\!\Downarrow \Leftrightarrow p\!\Downarrow$ for all p. Finally T is *compositional* if for all Δ, all $D \in \mathit{Ctxt}(\Delta)$ and p with $\Delta \vdash p$, we have $T(D)[T(p)] = T(D[p])$. A simple but helpful observation is:

Proposition 2.1 (Adequacy, [17]). *If a translation T is compositional and convergence equivalent, then T is adequate.*

If only new primitives are added to a calculus $\mathcal{C}'$, then full abstraction follows from moderate assumptions:

Proposition 2.2 (Full abstraction for extensions, [17]). *Let $\mathcal{C}, \mathcal{C}'$ be two calculi, let $\iota : \mathcal{C}' \to \mathcal{C}$ (the embedding) and $T : \mathcal{C} \to \mathcal{C}'$ be compositional and*

$$
\begin{array}{ll}
(\text{FUT.DEREF}(\mathsf{a})) & C[x] \mid x{\Leftarrow}v \to C[v] \mid x{\Leftarrow}v \\
(\beta\text{-CBV}(\mathsf{a})) & C[(\lambda x.e)\ v] \ \to\ C[e[v/x]] \\
(\text{CASE.BETA}(\mathsf{a})) & C[\mathbf{case}\ k_j(v_1,\ldots,v_{ar(k_j)})\ \mathbf{of}\ (k_i(x_1,\ldots,x_{ar(k_i)}) \Rightarrow e_i)^{i=1\ldots n}] \\
 & \qquad \to C[e_j[v_1/x_1,\ldots,v_{ar(k_j)}/x_{ar(k_j)}]] \\
(\text{CELL.DEREF}) & p \mid y\,\mathsf{c}\,x \mid x{\Leftarrow}v \to\ p \mid y\,\mathsf{c}\,v \mid x{\Leftarrow}v \\
(\text{GC}) & p \mid (\nu y_1)\ldots(\nu y_n)p' \ \to\ p \qquad \text{if } p' \text{ is successful and} \\
 & y_1,\ldots,y_n \text{ contain all process variables of } p' \\
(\text{DET.EXCH}) & (\nu x)(y{\Leftarrow}\widetilde{E}[\mathbf{exch}(x,v_1)] \mid x\,\mathsf{c}\,v_2) \to\ (\nu x)(y{\Leftarrow}\widetilde{E}[v_2] \mid x\,\mathsf{c}\,v_1)
\end{array}
$$

No capturing. We assume that no variables are moved out of their scope or into the scope of some other binder, i.e., $fv(v) \cap bv(C) = \emptyset$, and that processes are α-renamed when necessary.

Closure. Transformations are closed under $\equiv$ and process contexts.

Fig. 7. Correct transformation rules for $\lambda^{\tau}(\mathsf{fc})$

convergence equivalent translations, such that $T \circ \iota$ is the identity on $\mathcal{C}'$-programs, on $\mathcal{C}'$-observers, and on $\mathcal{C}'$-types. Then T as well as ι are fully abstract.

Correctness of Transformations in $\lambda^{\tau}(\mathsf{fc})$. In the correctness proofs we often rely on program transformations, which therefore must be sound for equivalence.

Definition 2.3. *A* program transformation $\approx$ *is a binary relation on processes. We say that $\approx$ is* correct *on (typed) processes if for all Δ and p, p' such that $\Delta \vdash p$ and $\Delta \vdash p'$ it holds that $p \approx p' \Rightarrow \Delta \vdash p \sim p'$.*

The use of the framework sketched above makes it possible to lift program equivalences from the untyped lambda calculus with futures [5] to correct program transformations in $\lambda^{\tau}(\mathsf{fc})$.

Theorem 2.4 (Correct transformations). *All of the following are correct transformations for $\lambda^{\tau}(\mathsf{fc})$:*

- *all reduction rules of $\lambda^{\tau}(\mathsf{fc})$, except for* CELL.EXCH($\mathsf{ev}$)
- *the transformations given in Fig. 7 (note the use of arbitrary contexts in the first three).*

The proof of this theorem proceeds by considering the untyped calculus $\lambda(\mathsf{f})$ that has no constructors and no case expressions. One defines an adequate translation $\lambda^{\tau}(\mathsf{fc}) \to \lambda(\mathsf{f})$ that allows us to transfer known equivalences from $\lambda(\mathsf{f})$ (see [5] and Appendix A) to $\lambda^{\tau}(\mathsf{fc})$. The translation $\lambda^{\tau}(\mathsf{fc}) \to \lambda(\mathsf{f})$ is factored through an intermediate calculus: first the arguments of data constructors are restricted to values, next case-expressions and constructors are encoded using abstractions, and types are removed. The complete proofs can be found in Appendix B.

$\tau \in \mathit{Type} ::= \mathsf{buf}\ \tau \mid \ldots$
$c \in \mathit{Const} ::= \mathbf{newBuf} \mid \mathbf{get} \mid \ldots$
$e \in \mathit{Exp} ::= \mathbf{put}(e_1, e_2) \mid \ldots$
$p \in \mathit{Proc} ::= x\,\mathsf{b}\,- \mid x\,\mathsf{b}\,v \mid \ldots$

ECs $\widetilde{E} ::= \mathbf{put}(\widetilde{E}, e) \mid \mathbf{put}(v, \widetilde{E}) \mid \ldots$
Future ECs $\widetilde{F} ::= \widetilde{E}[\mathbf{put}([\,], v)] \mid \widetilde{E}[\mathbf{get}\,[\,]] \mid \ldots$

Fig. 9. Extended evaluation contexts of $\lambda^\tau(\mathsf{fcb})$

Fig. 8. Syntactic extensions for $\lambda^\tau(\mathsf{fcb})$

$$
\begin{array}{lll}
(\textsc{buff.new}(\mathsf{ev})) & E[\mathbf{newBuf}\ v] & \to (\nu x)(E[x] \mid x\,\mathsf{b}\,-) \quad \text{fresh } x \\
(\textsc{buff.put}(\mathsf{ev})) & E[\mathbf{put}(x, v)] \mid x\,\mathsf{b}\,- & \longrightarrow E[\mathbf{unit}] \mid x\,\mathsf{b}\,v \\
(\textsc{buff.get}(\mathsf{ev})) & E[\mathbf{get}\ x] \mid x\,\mathsf{b}\,v & \to E[v] \mid x\,\mathsf{b}\,-
\end{array}
$$

Fig. 10. Extension of the reduction rules of $\lambda^\tau(\mathsf{fc})$ for $\lambda^\tau(\mathsf{fcb})$

$$
\mathbf{newBuf} : \alpha \to \mathsf{buf}\ \alpha \qquad \mathbf{get} : \mathsf{buf}\ \alpha \to \alpha
$$

$$
\frac{\Gamma \vdash e_1 : \mathsf{buf}\ \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathbf{put}(e_1, e_2) : \mathsf{unit}} \qquad \frac{x \notin \mathit{dom}(\Gamma)}{\Gamma \vdash x\,\mathsf{b}\,- : (x : \mathsf{buf}\ \tau)} \qquad \frac{\Gamma, x{:}\mathsf{buf}\ \tau \vdash v : \tau}{\Gamma \vdash x\,\mathsf{b}\,v : (x : \mathsf{buf}\ \tau)}
$$

Fig. 11. Typing of buffer-expressions and processes

3 Implementing Buffers

Any concrete realization of buffers will rely on (more or less intricate) non-interference properties and the preservation of various invariants. In this section, by extending the syntax and operational semantics of $\lambda^\tau(\mathsf{fc})$, we first provide a *specification* of one-place buffers that describes their desired behavior. We call the resulting calculus $\lambda^\tau(\mathsf{fcb})$. We then provide an implementation of buffers in $\lambda^\tau(\mathsf{fc})$. This induces a translation from $\lambda^\tau(\mathsf{fcb})$ into $\lambda^\tau(\mathsf{fc})$.

Specification. The calculus $\lambda^\tau(\mathsf{fc})$ is extended by new primitives for concurrent buffers. This defines the calculus $\lambda^\tau(\mathsf{fcb})$, with the syntactic extensions shown in Fig. 8. $\lambda^\tau(\mathsf{fcb})$ has two new components: $x\,\mathsf{b}\,-$ which represents an empty buffer, and $x\,\mathsf{b}\,v$ which represents a buffer that contains the value v. There are two new constants: **newBuf** to create a new buffer, and **get** to obtain the contents of a non-empty buffer (and emptying the buffer). There is also a new binary operator **put**, to place a new value into an empty buffer.

Fig. 10 summarizes the operational interpretation of the new constructs, and Fig. 9 extends the set of (future) evaluation contexts. Note that the reduction rules entail that **get** x suspends on an empty buffer x while $\mathbf{put}(x, v)$ suspends on a non-empty x. For typing we assume a new type constructor buf of arity 1. The typing of the constants is given by type schemes (see Fig. 11). Type safety then extends to the calculus $\lambda^\tau(\mathsf{fcb})$. Contextual preorder is defined as expected, where a *successful process* in $\lambda^\tau(\mathsf{fc})$ is extended such that $\lambda^\tau(\mathsf{fcb})$ allows $x\,\mathsf{b}\,-$ and $x\,\mathsf{b}\,v$ as additional components of successful processes.

$$
\begin{array}{l}
\mathit{newBuf} \triangleq \lambda\_.\,\mathsf{let}\ \langle h, f\rangle = \mathsf{newhandled}, \langle h', f'\rangle = \mathsf{newhandled},\\
\qquad\qquad \mathit{putg} = \mathbf{cell}(\mathsf{True}), \mathit{getg} = \mathbf{cell}(f),\\
\qquad\qquad \mathit{stored} = \mathbf{cell}(f'), \mathit{handler} = \mathbf{cell}(h)\\
(1)\qquad \mathsf{in}\ \mathbf{thread}\ \lambda\_.\langle \mathit{putg}, \mathit{getg}, \mathit{stored}, \mathit{handler}\rangle\ \mathsf{end}
\end{array}
$$

$$
\begin{array}{l}
\mathit{put} \triangleq \lambda\langle\langle x_p, x_g, x_s, x_h\rangle, v\rangle.\\
\qquad \mathsf{let}\ \langle h, f\rangle = \mathsf{newhandled}\\
(1)\qquad \mathsf{in\ wait}\ (\mathbf{exch}(x_p, f));\\
(2)\qquad\quad \mathbf{exch}(x_s, v);\\
(3)\qquad\quad (\mathbf{exch}(x_h, h))(\mathsf{True})\\
\qquad \mathsf{end}
\end{array}
\qquad
\begin{array}{l}
\mathit{get} \triangleq \lambda\langle x_p, x_g, x_s, x_h\rangle.\\
\qquad \mathsf{let}\ \langle h, f\rangle = \mathsf{newhandled}\\
\qquad\quad \langle h', f'\rangle = \mathsf{newhandled}\\
(1)\qquad \mathsf{in\ wait}\ (\mathbf{exch}(x_g, f));\\
(2)\qquad\quad \mathsf{let}\ v = (\mathbf{exch}(x_s, f'))\\
(3)\qquad\quad \mathsf{in}\ (\mathbf{exch}(x_h, h))(\mathsf{True});\ v\ \mathsf{end}\\
\qquad \mathsf{end}
\end{array}
$$

Fig. 12. Implementing the buffer operations *newBuf*, *put* and *get*. The numbers (1), (2), (3) indicate subexpressions for later reference.

Implementation. A particular implementation of buffers in terms of reference cells and futures is considered.

From now on we assume that the set of type constructors contains a nullary constructor $\mathsf{bool} \in \mathcal{K}$, with nullary data constructors True and False. We will freely use the usual syntax sugar such as a (non-recursive) let-binding $\mathsf{let}\ x_1{=}e_1, \ldots, x_n{=}e_n\ \mathsf{in}\ e\ \mathsf{end}$ and sequencing $e_1; e_2$, and also use patterns in abstractions $\lambda\pi.e$ as shorthand for $\lambda x.\mathbf{case}\ x\ \mathbf{of}\ \pi \Rightarrow e$ etc. Instead of $\mathbf{case}\ e\ \mathbf{of}\ \mathsf{True} \Rightarrow e_1 \mid \mathsf{False} \Rightarrow e_2$ we write $\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$, and the special case $\mathsf{if}\ e\ \mathsf{then\ True\ else\ True}$ is written as $\mathsf{wait}\ e$. The symbol '_' stands for an arbitrary fresh variable. Finally, we write $\mathsf{newhandled}$ as shorthand for $\mathbf{handle}\ \lambda f h.\langle h, f\rangle$.

The implementation in $\lambda^{\tau}(\mathsf{fc})$ of operations corresponding to **newBuf**, **put**, and **get** is shown in Fig. 12. The buffer data structure is here implemented as a tuple, consisting of four reference cells:

$$\mathit{buf}\,\tau \ \triangleq\ \mathsf{ref\ bool} \times \mathsf{ref\ bool} \times \mathsf{ref}\ \tau \times \mathsf{ref}\ (\mathsf{bool} \to \mathsf{unit}).$$

The first and second of these reference cells serve as guards to ensure that succeeding *put* and *get* operations alternate. Exactly one of them will contain a handled future: if the first guard contains a future, this indicates that the buffer is currently non-empty, hence *put* must block. Likewise, if the second guard contains a handled future, the tuple represents an empty buffer and *get* must block. The final reference cell stores a handler for this future. The third cell, of type $\mathsf{ref}\ \tau$, stores the actual contents of the buffer. When representing an empty buffer, this reference will contain a handled future of type τ as a dummy value. In summary, there are the following invariants associated with the value $\langle \mathit{putg}, \mathit{getg}, \mathit{stored}, \mathit{handler}\rangle$ that implements the buffer:

- the guards *putg* and *getg* contain either a handled future or True (perhaps reachable via dereferencing futures),
- at most one of *putg* and *getg* contains True,

- whenever *getg* contains True then the value in *stored* is the value currently in the buffer, and
- whenever *putg* contains True then the value in *stored* is 'garbage', representing an empty buffer.

The procedure *newBuf* yields a tuple representing an empty buffer, satisfying the invariants. The procedure *put*, when applied to a buffer $\langle putg, getg, stored, handler\rangle$ and a value v, suspends until the buffer is guaranteed to be empty. This is achieved by pattern matching on the contents of *putg* (using wait): since the first argument position of the **case** construct constitutes a future EC, *put* can continue only when *putg* contains a proper (non-future) value. By the invariants, this implies that the buffer is empty. At the same time, *putg* is replaced by a fresh future f, with handle h, to indicate that the buffer will be non-empty after *put* succeeds. After writing v to the cell *stored*, the second guard *getg* is set to True (perhaps via a reference) to permit following *get* operations to succeed. This is done using the handle stored in the reference cell *handler*, which is replaced by the handle h for the freshly introduced future f. The procedure *get* is analogous (partly symmetric) to *put*.

The use of the handled futures in *put* and *get* is somewhat subtle: in general, several threads concurrently attempt to place values into the buffer (and dually, for reading from the buffer). The thread that is scheduled first replaces the contents of the guard by a future f_1. This future can be bound only after this instance of *put* has finished. A second instance of *put* can proceed immediately with its own exchange operation, replacing f_1 by a future f_2 before the wait suspends on f_1. In this way, a chain of threads suspending on futures $f_1, f_2, \ldots$ in their respective *put* operations can build up. At the same time, a chain of threads suspending in their respective *get* operations can build up.

Implementation as Translation. The implementation gives rise to a *translation* T_B from $\lambda^{\tau}(\mathsf{fcb})$ into $\lambda^{\tau}(\mathsf{fc})$: **put**, **get**, and **newBuf** are replaced by the resp. program code, *put*, *get*, and *newBuf* from Fig. 12, where for **put**, the two arguments are translated into a pair. On process level, we replace:

$$\begin{aligned} x\,\mathsf{b}\,- \mapsto\ & (\nu x_p)(\nu x_g)(\nu x_s)(\nu x_h)x{\Leftarrow}\langle x_p, x_g, x_s, x_h\rangle \\ & \mid (\nu h)(\nu f)(\nu h')(\nu f')(h\,\mathsf{h}\,f \mid h'\,\mathsf{h}\,f' \mid x_p\,\mathsf{c}\,\mathsf{True} \mid x_g\,\mathsf{c}\,f \mid x_s\,\mathsf{c}\,f' \mid x_h\,\mathsf{c}\,h) \\ x\,\mathsf{b}\,v\ \mapsto\ & (\nu x_p)(\nu x_g)(\nu x_s)(\nu x_h)x{\Leftarrow}\langle x_p, x_g, x_s, x_h\rangle \\ & \mid (\nu h)(\nu f)(h\,\mathsf{h}\,f \mid x_p\,\mathsf{c}\,f \mid x_g\,\mathsf{c}\,\mathsf{True} \mid x_s\,\mathsf{c}\,T_B(v) \mid x_h\,\mathsf{c}\,h). \end{aligned}$$

Formally, these replacements homomorphically extend to a mapping $T_B : \lambda^{\tau}(\mathsf{fcb}) \to \lambda^{\tau}(\mathsf{fc})$ on all $\lambda^{\tau}(\mathsf{fcb})$-terms, -processes, and -contexts. A corresponding type translation is defined inductively, by $T_B(\mathsf{buf}\ \tau) \triangleq buf\,(T_B\tau)$, and proceeding homomorphically in all other cases.

These mappings are compatible with typing: if $\Delta \vdash p$ then $T_B(\Delta) \vdash T_B(p)$, where $T_B(\Delta) \triangleq (T_B(\Gamma), T_B(\Gamma'))$ for $\Delta = (\Gamma, \Gamma')$ and $T_B(x_1{:}\tau_1, \ldots, x_n{:}\tau_n) \triangleq x_1{:}T_B(\tau_1), \ldots, x_n{:}T_B(\tau_n)$ denotes the pointwise extension to type environments. Corresponding typing properties hold for contexts, so that T_B forms a translation in the sense of [17]. It is easy to see that:

Lemma 3.1 (Compositionality). *The translation $T_B : \lambda^{\tau}(\mathsf{fcb}) \to \lambda^{\tau}(\mathsf{fc})$ is compositional, i.e., for all p, D, we have $T_B(D)[T_B(p)] = T_B(D[p])$.*

4 Correctness of Buffer Implementations

Convergence Equivalence. We argue that the buffer implementation, described by $T_B : \lambda^\tau(\mathsf{fcb}) \to \lambda^\tau(\mathsf{fc})$ above, is correct. To this end, we will prove T_B convergence equivalent in this section, and use compositionality (Lemma 3.1). By Lemma 2.1, this entails that T_B is adequate. Moreover, we will prove that $\lambda^\tau(\mathsf{fcb})$ is an extension of $\lambda^\tau(\mathsf{fc})$, which means $\leq$ is extended conservatively.

Lemma 4.1 (T_B preserves success). *Let p be a $\lambda^\tau(\mathsf{fcb})$-process.*

1. *If p is successful, then so is $T_B(p)$. In particular, $T_B(p)\Downarrow$ in this case.*
2. *If $T_B(p)$ is successful, then p is also a successful process.*

The definition of the translation T_B also shows the following.

Lemma 4.2 (Context translation). *If D is a process context of $\lambda^\tau(\mathsf{fcb})$, then $T_B(D)$ is a process context. If E is an evaluation context of $\lambda^\tau(\mathsf{fcb})$, then $T_B(E)$ is an evaluation context in $\lambda^\tau(\mathsf{fc})$.*

Note that the corresponding property does not hold for future evaluation contexts. For example, **get** [] forces its argument to be a buffer before reducing, while its image *get* [] proceeds with arbitrary values as argument. The argument ends in a future EC only after further beta- and case-reductions.

Proposition 4.3 ($\downarrow$-preservation). *For every $\lambda^\tau(\mathsf{fcb})$-process p, $p\downarrow \Rightarrow T_B(p)\downarrow$.*

Proof (sketch). The idea is to simulate reductions of $\lambda^\tau(\mathsf{fcb})$ in $\lambda^\tau(\mathsf{fc})$, up to contextual equivalence: for every $p_1 \xrightarrow{\mathsf{ev}} p_2$ there exists $p_1', p_2' \in \lambda^\tau(\mathsf{fc})$ such that $T_B(p_1) \sim p_1' \xrightarrow{\mathsf{ev},*} p_2' \sim T_B(p_2)$. The equivalences are established by transformations from Theorem 2.4. Then, we can use induction on the length of a successful reduction sequence in $\lambda^\tau(\mathsf{fcb})$ to derive a successful sequence for the translated process. Proposition C.3 in the appendix gives the detailed proof. □

The converse implication requires a more careful analysis. The execution of each instance of *put* or *get* (*newBuf* provides no problems) consists of initial (β-CBV(ev)) and (CASE.BETA(ev)) reductions, and eventually the argument has to be evaluated in a future-strict context. The ensuing (CASE.BETA(ev)) and (FUT.DEREF(ev))-reduction (pattern matching on the cells in a tuple $\langle p, g, s, h\rangle$ and proceeding after wait, resp.) can be ignored in the following analysis. Referring to Fig. 12, the internal code-pointer is denoted 1a, 1b, 2, 3a, 3b. For instance, we describe subexpressions of *put* as follows: (1) for $\mathsf{wait}\ (\mathbf{exch}(x_p, f))$ with (1a) for $\mathbf{exch}\ldots$ and (1b) for wait, (2) for $\mathbf{exch}(x_s, v_i)$, and (3) for $(\mathbf{exch}(x_h, h_{p,i}))(\mathsf{True})$ with (3a) for $\mathbf{exch}\ldots$ and (3b) for handle binding. The subexpressions of *get* are described similarly.

One of the difficulties is illustrated by a process $x \Leftarrow \mathbf{put}(y, v) \mid y \Leftarrow \mathbf{get}\ x \mid \ldots$. Assume that **get** is executed first, then **put**. For the corresponding reduction sequence in $\lambda^\tau(\mathsf{fc})$ it is unavoidable that essentially same sequence is used on the implementation *get* and *put*. However, the initial reductions of *put* may be

executed earlier. (In the case of $y \overset{susp}{\Longleftarrow} get\ x$, this is even enforced.) For the reduction in the implementation, this means that the reduction steps of instances of *get* and *put* cannot be gathered into one contiguous block; this is possible only for the main steps 1,2,3 of an instance.

Due to space limitations, the detailed analysis of the possible behavior of the implementation is found in Appendix C. It implies the following sequencing constraint of reductions, where $\xrightarrow{x,a,b}$ means a reduction step for a particular buffer x of the *put*/*get*-instance b with code-pointer a.

Lemma 4.4. *For a buffer x the following sequencing holds in a $\xrightarrow{\mathsf{ev}}$-reduction: If for two instances b_1, b_2 of put, get: $\xrightarrow{x,a,b_1}$ is before $\xrightarrow{x,a,b_2}$ for some $a \in \{1b, 2, 3a, 3b\}$, then for all $a_1, a_2 \in \{1b, 2, 3a, 3b\}$: $\xrightarrow{x,a_1,b_1}$ is before $\xrightarrow{x,a_2,b_2}$. For two put-instances b_1, b_2 (or get-instances, respectively): $\xrightarrow{x,1a,b_1}$ is before $\xrightarrow{x,1a,b_2}$ iff $\xrightarrow{x,2,b_1}$ is before $\xrightarrow{x,2,b_2}$.*

Proposition 4.5 (↓-reflection). *For every $\lambda^\tau(\mathsf{fcb})$-process p, $T_B(p)\downarrow \Rightarrow p\downarrow$.*

Proof (sketch; the full proof appears in the appendix as Prop. C.4). The main idea of the proof is to rearrange a reduction sequence R of $T_B(p)$ to a successful process p_ω, such that it can be retranslated into $\lambda^\tau(\mathsf{fcb})$. One obstacle is mentioned above: the initial reduction steps and the steps 1,2,3 have to be treated separately. It is possible to rearrange the reduction steps 1,2,3 such that the reduction steps of every instance of *put*,*get* occur without interrupt. The strategy is to move the reductions (1b) and then (1a) close to their corresponding (2)-reduction, starting from the rightmost reduction step. Next reductions from (3a), and then (3b), are moved to the left to their corresponding (2)-reduction, starting from the left. These commutations are justified by Lemma 4.4. The initial reduction steps, including dereferencing of the buffer variable, are correct transformations due to Theorem 2.4. After this reordering of R a retranslation is possible, which shows $p\downarrow$. □

Proposition 4.6 (⇓-reflection). *For every $\lambda^\tau(\mathsf{fcb})$-process p, $T_B(p)\Downarrow \Rightarrow p\Downarrow$.*

Proof. Suppose that for the $\lambda^\tau(\mathsf{fcb})$-process p we have $p\uparrow$. We show $T_B(p)\uparrow$. Since $p\uparrow$ there is a reduction R from p to a process $p_0\Uparrow$. Analogous to the proof of Proposition 4.3, we can show by induction on the length of R that there is a sequence R' of correct transformations and reductions from $T_B(p)$ to the process $T_B(p_0)$. Proposition 4.5 applied to p_0 shows that $T_B(p_0)\downarrow$ is impossible, hence $T_B(p_0)\Uparrow$ holds. By induction on the length of R' (which consists of ev-reductions and correct transformations), Corollary B.20 is used to show $T_B(p)\uparrow$. □

The proof of the following proposition is more intricate:

Proposition 4.7 (⇓-preservation). *For every $\lambda^\tau(\mathsf{fcb})$-process p, $p\Downarrow \Rightarrow T_B(p)\Downarrow$.*

Proof. The detailed proof is in Appendix C; here we give a sketch. We prove the equivalent claim that for every $\lambda^\tau(\mathsf{fcb})$-process p, $T_B(p)\uparrow \Rightarrow p\uparrow$. As in the

proof of Proposition 4.5, a given reduction R corresponding to $T_B(p)\uparrow$ will be rearranged and modified in order to construct a $\lambda^\tau(\mathsf{fcb})$-reduction of p that shows $p\uparrow$. This allows a similar rearrangement into 1,2,3-blocks and intermediate correct transformations. However, this is not possible for all instances of *put get*, since some of these may be started but never completed in the reduction.

Variants of the following argument can be used to overcome this difficulty. Suppose a certain instance m of *put* has been started within R, but the next reduction step, say from (3a), is missing in R. Let p_ω be the last process in R, for which we necessarily have $p_\omega\Uparrow$. The commutation properties show that $\xrightarrow{3a,m}$ is a reduction possibility of p_ω, i.e. $p_\omega \xrightarrow{3a,m} p_{\omega,1}$, which immediately implies $p_{\omega,1}\Uparrow$. Thus we extend R to $R\cdot \xrightarrow{3a,m}$. This procedure is repeated until all partially executed instances are completed; as an invariant, the number of started instances of *put*, *get* is not increased. Now it is possible to construct a reduction sequence showing $p\uparrow$. Finally, there is some process p_ω with $p \xrightarrow{*} p_\omega$ and $T_B(p_\omega) = q_\omega$, where q_ω is the final process of the rearranged and extended sequence R'. The property $q_\omega\Uparrow$ can be shown using Theorem 2.4. Lemma 4.3 shows that $p_\omega\downarrow$ is impossible, hence $p_\omega\Uparrow$. This implies $p\uparrow$. □

Full Abstraction. Propositions 4.3, 4.5, 4.6, 4.7, and 2.1 imply:

Theorem 4.8 (Adequacy). *The translation T_B: $\lambda^\tau(\mathsf{fcb}) \to \lambda^\tau(\mathsf{fc})$ is adequate.*

We need also correctness of several transformations in $\lambda^\tau(\mathsf{fcb})$, which can be shown using adequacy of the translation T_B. (The detailed proof is given in the appendix in Lemma C.6).

Theorem 4.9 (Correct transformations in $\lambda^\tau(\mathsf{fcb})$). *The following holds:*

- *All reduction rules of $\lambda^\tau(\mathsf{fcb})$ except for* CELL.EXCH(ev), BUFF.GET(ev), *and* BUFF.PUT(ev) *are correct.*
- *The transformations β-*CBV(a), FUT.DEREF(a), CELL.DEREF, GC *and* DET.EXCH *(see Fig. 7) lifted to $\lambda^\tau(\mathsf{fcb})$ are correct.*

Theorem 4.10 (Full abstraction). *$T_B : \lambda^\tau(\mathsf{fcb}) \to \lambda^\tau(\mathsf{fc})$ and the identity $\lambda^\tau(\mathsf{fc}) \to \lambda^\tau(\mathsf{fcb})$ are fully abstract.*

Proof. This follows from adequacy (Theorem 4.8) and Proposition 2.2, since the identity translation $\lambda^\tau(\mathsf{fc}) \to \lambda^\tau(\mathsf{fcb})$ is an embedding of $\lambda^\tau(\mathsf{fc})$ into $\lambda^\tau(\mathsf{fcb})$. □

Some Axiomatic Laws for Buffers. A common approach to the specification of abstract data types, in the sequential case, is by an axiomatic description of the operations. The machinery developed above allows us to prove that the buffers of $\lambda^\tau(\mathsf{fc})$ satisfy such axioms. Using adequacy of T_B (Theorem 4.8), the implied correctness of transformations for $\lambda^\tau(\mathsf{fcb})$ (Theorem 4.9), and correctness of program transformations for $\lambda^\tau(\mathsf{fc})$ (Theorem 2.4), one can show the following rules for **put** and **get** are correct:

$$\begin{array}{ll} (\text{DET.PUT}) & (\nu x).E[\mathbf{put}(x,v)] \mid x\,\mathsf{b}\,- \to (\nu x).E[\mathbf{unit}] \mid x\,\mathsf{b}\,v \\ (\text{DET.GET}) & (\nu x).E[\mathbf{get}\ x] \mid x\,\mathsf{b}\,v \to (\nu x).E[v] \mid x\,\mathsf{b}\,- \end{array}$$

The detailed proof can be found in the appendix (Proposition C.7). Note that these equivalences are like BUFF.PUT(ev) and BUFF.GET(ev), but restricted to sequentially used buffers. Then,

$$\Delta \vdash E[\mathbf{get}(\mathbf{put}(\mathbf{newBuf}\ \mathbf{unit}, v))] \sim E[v]$$

and similar equivalences follow.

5 Conclusion

We have proved formally that concurrent buffers can be correctly expressed in the lambda calculus with futures. This illustrates how recent proof techniques based on observational semantics permit to prove for a first time the equivalence of various concurrency primitives of realistic concurrent programming languages. In future work, it remains to elaborate this statement further, by proving the equivalence of concurrent buffers, channels, and handled futures in the context of the lambda calculus with futures. In particular, we have to show that we can remove handles from $\lambda^{\tau}(\mathsf{fcb})$ by some correct encoding, and that we can encode channels in this calculus too. Proving similar results for other base languages is possible in principle, but requires to establish a sufficiently rich equational theory first.

References

1. A. Carayol, D. Hirschkoff, and D. Sangiorgi. On the representation of McCarthy's amb in the pi-calculus. *TCS*, 330(3):439–473, 2005.
2. R. De Nicola and M. Hennessy. Testing equivalences for processes. *TCS*, 34:83–133, 1984.
3. C. Fournet, F. Le Fessant, L. Maranget, and A. Schmitt. JoCaml: A language for concurrent distributed and mobile programming. In *Advanced Functional Programming*, volume 2638 of *LNCS*, 129–158. 2002.
4. S. P. Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *POPL*, 295–308. ACM Press, 1996.
5. J. Niehren, D. Sabel, M. Schmidt-Schauß, and J. Schwinghammer. Observational semantics for a concurrent lambda calculus with reference cells and futures. In *23rd MFPS*, *ENTCS* 173, 313–337. 2007.
6. J. Niehren, J. Schwinghammer, and Gert Smolka. A concurrent lambda calculus with futures. *TCS*, 364(3):338–356, 2006.
7. C.-H. L. Ong. Non-determinism in a functional setting. In *LICS '93*, 275–286. IEEE Computer Society, 1993.
8. M.J. Parkinson, R. Bornat, and P.W. O'Hearn. Modular verification of a non-blocking stack. In *POPL*, 297–302. ACM Press, 2007.
9. J. Parrow. Expressiveness of process algebras. *ENTCS*, 209:173–186, 2008.
10. S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *POPL*, 295–308, 1996. ACM Press.
11. J.H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

12. J.G. Riecke. Fully abstract translations between functional languages. In *POPL*, 245–254, 1991. ACM-Press.
13. E. Ritter and A.M. Pitts. A fully abstract translation between a lambda-calculus with reference types and standard ML. In *2nd TLCA*, *LNCS* 902, 397–413, 1995.
14. A. Rossberg, D. Le Botlan, G. Tack, T. Brunklaus, and G. Smolka. *Trends in Functional Programming*, vol. 5, Alice Through the Looking Glass, 79–96. 2006.
15. D. Sabel and M. Schmidt-Schauß. A call-by-need lambda calculus with locally bottom-avoiding choice: context lemma and correctness of transformations. *Mathematical Structures in Computer Science*, 18(3):501–553, 2008.
16. M. Schmidt-Schauß and D. Sabel. On generic context lemmas for lambda calculi with sharing. Frank report 27, Inst. Informatik, Goethe-Univ., Frankfurt, 2007.
17. M. Schmidt-Schauß, J. Niehren, D. Sabel, and J. Schwinghammer. Adequacy of compositional translations for observational semantics. In *5th IFIP International Conference on TCS*, volume 273 of IFIP, 521–535. 2008.

A Modifying β-cbv$_{\mathrm{L}}$(ev)-reduction in $\lambda(\mathsf{f}')$

The calculus $\lambda(\mathsf{f}')$ described in [5] is the subcalculus of $\lambda^{\tau}(\mathsf{fc})$ without constructors, case-expressions, and typing. Moreover, instead of the β-CBV(ev)-reduction rule from Fig. 4, a sharing variant used that introduces a new future instead of substituting into the body. This 'lazy' call-by-value β-rule (β-CBV$_{\mathrm{L}}$(ev)) takes the form $E[(\lambda y.e)\, v] \to (\nu y)(E[e] \mid y{\Leftarrow}v)$.

We call the modification of $\lambda(\mathsf{f}')$ with the usual (substituting) call-by-value β-reduction $\lambda(\mathsf{f})$ (this is the calculus presented in [6]). We first prove that the identity translation from $\lambda(\mathsf{f})$ into $\lambda(\mathsf{f}')$ is fully-abstract. This essentially means that, as far as contextual equivalence is concerned, it does not matter which β-rule we choose.

For the proof we expect familiarity with the definitions, lemmas and theorems in [5]. Obviously, the identity translation from $\lambda(\mathsf{f}')$ to $\lambda(\mathsf{f})$ as well as the identity translation from $\lambda(\mathsf{f})$ to $\lambda(\mathsf{f}')$ are bijective and compositional. We show that both translations are convergence equivalent, which by Proposition 2.2 implies that both translations are fully-abstract.

Lemma A.1. *For all $\lambda(\mathsf{f}')$-processes p, $p{\downarrow} \Leftrightarrow p{\downarrow_{\lambda(\mathsf{f})}}$ and $p{\Downarrow} \Leftrightarrow p{\Downarrow_{\lambda(\mathsf{f})}}$, i.e. the identity translations from $\lambda(\mathsf{f}')$ and $\lambda(\mathsf{f})$ and back are convergence equivalent.*

Proof. 1. Theorem 4.23 of [5] proves that β-CBV(a) is a correct program transformation for $\lambda(\mathsf{f}')$. From $p{\downarrow_{\lambda(\mathsf{f})}}$, induction on the length of a reduction from p to a successful process shows that $p{\downarrow}$. This is because the reduction corresponding to $p{\downarrow_{\lambda(\mathsf{f})}}$ consists of $\lambda(\mathsf{f}')$-reductions and β-CBV(ev)-reductions.

2. In order to prove the other direction, assume that $p{\downarrow}$, i.e. $p \xrightarrow{k} p'$ where p' is a successful process of $\lambda(\mathsf{f}')$. We use induction on k. If $k = 0$, then p is also a successful process of $\lambda(\mathsf{f})$. For the induction step there are two cases: If the first reduction of $p \to p'' \xrightarrow{k-1} p'$ is also a reduction of $\lambda(\mathsf{f})$, then the claim follows by using the induction hypothesis. If the reduction is a β-CBV$_{\mathrm{L}}$(ev)-reduction, then we have the following situation:

$$\begin{array}{ccc} p = D[E[(\lambda y.e)\ v]] & \xrightarrow{\beta\text{-CBV(ev)}} & \overline{p} = D[E[e[v/y]]] \\ {\scriptstyle \beta\text{-CBV}_{\mathrm{L}}(\mathsf{ev})}\downarrow & \nearrow_{(\text{FUT.DEREF}(\mathsf{a})\ \vee\ \text{GC}),*} & \\ p'' = D[E[e] \mid y{\Leftarrow}v] & & \\ {\scriptstyle \mathsf{ev},k-1}\downarrow & & \\ p' & & \end{array}$$

It is easy to verify that the sequence of (FUT.DEREF(a) $\vee$ GC)-transformations always exists. It is sufficient to show that there exists a reduction from $\overline{p}$ to a successful process of length less than k. This implies that we can apply the induction hypothesis to $\overline{p}$ and the claim follows. The missing part follows from Lemma 4.18 (1) and the proof of Proposition 4.31 in [5] for the transformation (FUT.DEREF(a)); for (GC) it follows from Lemma 4.7 in combination with Theorem 4.8 in [5].

3. Assume that $p{\uparrow_{\lambda(\mathsf{f})}}$, and note that (1),(2) imply $p{\Uparrow} \Leftrightarrow p{\Uparrow_{\lambda(\mathsf{f})}}$. Induction on the length of a reduction shows that $p{\uparrow}$, where the induction step is either obvious or follows from correctness of the transformation β-CBV(a).
4. To show the last case assume that $p{\uparrow}$. The proof is by induction on the length of a reduction to a must-divergent process, using the same methods as (2). □

Theorem A.2. *The identity translations from $\lambda(\mathsf{f}')$ into $\lambda(\mathsf{f})$ and back are fully-abstract.*

This result immediately allows us to transfer correct program transformations for $\lambda(\mathsf{f}')$ (shown in [5]) to $\lambda(\mathsf{f})$.

Corollary A.3. *All reductions of $\lambda(\mathsf{f})$ except* CELL.EXCH(ev)*, and all the transformations β-*CBV(a)*,* FUT.DEREF(a)*,* CELL.DEREF*,* GC *and* DET.EXCH *(see Fig. 7) are correct program transformations for $\lambda(\mathsf{f})$.*

Proof. Correctness of the transformations was established for $\lambda(\mathsf{f}')$ in [5]. Because of full abstraction, they are also correct in $\lambda(\mathsf{f})$. □

B Removing Constructors and Case-Expressions

The constructors and case-expressions are removed from $\lambda^{\tau}(\mathsf{fc})$ in two steps. We first provide an encoding from $\lambda^{\tau}(\mathsf{fc})$ into a subset of itself. Let $\lambda^{\tau}(\mathsf{fc}_{av})$ be the sublanguage of $\lambda^{\tau}(\mathsf{fc})$ where in constructor applications $k(e_1,\ldots,e_n)$ the arguments e_i are restricted to values.

We provide an encoding $enc_1 : \lambda^{\tau}(\mathsf{fc}) \to \lambda^{\tau}(\mathsf{fc}_{av})$, where additionally we introduce some labels to mark the encoded constructors. These labels are used in later proofs. The encoding enc_1 is defined as follows

$$enc_1(k(e_1,\ldots,e_n)) \triangleq (\lambda^{l_1}x_1.\ \ldots\lambda^{l_n}x_n.k(x_1,\ldots,x_n))\ enc_1(e_1)^{l_1}\ldots enc_1(e_n)^{l_n},$$
where x_i are fresh, and l_i are new labels
$$enc_1(t) \triangleq \text{homomorphically wrt. the term structure of } t$$

extended to contexts in the evident way, and acting as identity on types.

The second step is an encoding enc_2 that maps processes of $\lambda^{\tau}(\mathsf{fc}_{av})$ to $\lambda(\mathsf{f})$-processes, by removing constructors, case expressions, and types: Let $K = \mathcal{D}(\kappa(\tau_1,\ldots,\tau_m))$ be the set of constructors for a specific type constructor κ and types $\tau_1,\ldots,\tau_m$. By the assumptions on the signature, K is non-empty. We choose an arbitrary (but from now fixed) order of the constructors in K, $k_1,\ldots,k_n$ where $n \geq 1$. A constructor application of $\lambda^{\tau}(\mathsf{fc}_{av})$ is encoded as:

$$enc_2(k_i(v_1,\ldots,v_{ar(v_i)})) \triangleq (\lambda p_1,\ldots,p_n.p_i\ enc_2(v_1)\ \ldots\ enc_2(v_{ar(k_i)})\ \mathbf{unit})$$

The additional **unit** argument to p_i achieves the correct behavior in the case of nullary constructors, with respect to call-by-value semantics. The encoding for **case** expressions is the following:

$$enc_2(\mathbf{case}_{\kappa(\tau_1,\ldots,\tau_m)}\ e\ \mathbf{of}\ (k_i(x_{i,1},\ldots,x_{i,ar(k_i)}) \Rightarrow e_i{}^{i=1\ldots n})) \triangleq$$
$$enc_2(e)\ (\lambda x_{1,1},\ldots,x_{1,ar(k_1)}.\lambda\_.enc_2(e_1))\ldots(\lambda x_{n,1},\ldots,x_{n,ar(k_n)}.\lambda\_.enc_2(e_n))$$

Again, the additional abstraction $\lambda\_.\ldots$ in each branch leads to a uniform translation, including the correct behavior for nullary constructors. For all other constructs, the encoding enc_2 operates homomorphically wrt. the term structure. Types are removed by this second encoding.

Definition B.1. *The translation* $enc : \lambda^\tau(\mathsf{fc}) \to \lambda(\mathsf{f})$ *is the composition* $enc_2 \circ enc_1$ *of the above translations* enc_1 *and* enc_2.

As an example we demonstrate the encoding of lists (over a fixed element type τ). Assume that $\mathcal{D}$ contains constructors $\mathbf{cons} : \tau \to \mathit{list} \to \mathit{list}$ and $\mathbf{nil} : \mathit{list}$. Then:

$$\begin{aligned} enc(\mathbf{cons}(e_h, e_t)) &= enc_2(enc_1(\mathbf{cons}(e_h, e_t))) \\ &= enc_2(\lambda x_1, x_2\ (\mathbf{cons}(x_1, x_2))\ enc_1(e_h)\ enc_1(e_t)) \\ &= (\lambda x_1, x_2, a_{nil}, a_{cons}.x_1\ x_2\ \mathbf{unit})\ enc(e_h)\ enc(e_t) \\ enc(\mathbf{nil}) &= enc_2(\mathbf{nil}) = (\lambda a_{nil}, a_{cons}.a_{nil}\ \mathbf{unit}) \end{aligned}$$

$$enc\begin{pmatrix}\mathbf{case}_{list}\ e\ \mathbf{of} \\ \quad \mathbf{nil} \Rightarrow e_1 \\ \quad \mid \mathbf{cons}(x, y) \Rightarrow e_2\end{pmatrix} = enc(e)\ (\lambda\_.enc(e_1))\ (\lambda x, y.\lambda\_.enc(e_2))$$

B.1 Adequacy of enc_2

Lemma B.2. *For every value* v *of* $\lambda^\tau(\mathsf{fc}_{av})$, $enc_2(v)$ *is a* $\lambda(\mathsf{f})$*-value.*

Proof. Clear, since all values are translated into abstractions. □

Lemma B.3. *For all* $p_1, p_2 \in \lambda^\tau(\mathsf{fc}_{av})$ *with* $p_1 \xrightarrow{\mathsf{ev}} p_2$, $enc_2(p_1) \xrightarrow{\mathsf{ev},*} enc_2(p_2)$.

Proof. First observe that the EC or future EC, resp., containing the redex that is contracted by $p_1 \xrightarrow{\mathsf{ev}} p_2$ must correspond to an EC or future EC, resp., for $\lambda(\mathsf{f})$: since only $\lambda^\tau(\mathsf{fc}_{av})$-values appear in constructor applications, the hole cannot be below a constructor. Hence the only exceptional case is the decomposition $p_1 = D[E[\mathbf{case}\ E'\ \ldots]]$, but in this case the translation wrt. enc_2 does yield an EC. Now one needs to check that the reduction $p_1 \xrightarrow{\mathsf{ev}} p_2$ is transferable. The only exception is a CASE.BETA(ev)-reduction, and we prove this case explicitly:

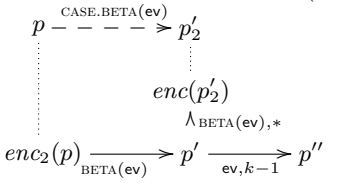
$$\begin{aligned} &enc_2(p_1) \\ &= enc_2(D[E[\mathbf{case}\ k_j(v_1, \ldots, v_{ar(k_j)})\ \mathbf{of}\ (k_i(x_1, \ldots, x_{ar(k_i)}) \Rightarrow e_i)^{i=1\ldots n}]]) \\ &= enc_2(D)[enc_2(E)[enc_2(\mathbf{case}\ k_j(v_1, \ldots, v_{ar(k_j)})\ \mathbf{of}\ (k_i(x_1, \ldots, x_{ar(k_i)}) \Rightarrow e_i)^{i=1\ldots n}])] \\ &= enc_2(D)[enc_2(E)\left[\begin{array}{l}((\lambda p_1, \ldots, p_n.p_j\ enc_2(v_1)\ \ldots\ enc_2(v_{ar(k_j)})\ \mathbf{unit}) \\ \quad (\lambda x_{1,1}, \ldots, x_{1,ar(k_1)}.\lambda\_.enc_2(e_1)) \\ \quad \ldots \\ \quad (\lambda x_{n,1}, \ldots, x_{n,ar(k_n)}.\lambda\_.enc_2(e_n))\end{array}\right])] \\ &\xrightarrow{\beta\text{-CBV}(\mathsf{ev}),*} \\ &\quad enc_2(D)[enc_2(E)[((\lambda x_{j,1}, \ldots, x_{j,ar(k_j)}.\lambda\_.enc_2(e_j))\ enc_2(v_1) \ldots enc_2(v_{ar(k_j)})\ \mathbf{unit})]] \\ &\xrightarrow{\beta\text{-CBV}(\mathsf{ev}),*} enc_2(D)[enc_2(E)[enc_2(e_j)[enc_2(v_1)/x_{j,1}, \ldots, enc_2(v_{ar(k_j)})/x_{j,ar(k_j)}]]] \\ &= enc_2(p_2) \end{aligned}$$

Note,that Lemma B.2 ensures that all $enc_2(v_i)$ are values for $\lambda(\mathsf{f})$, validating the β-CBV(ev) steps. □

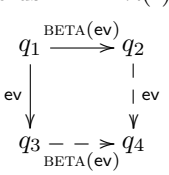
Proposition B.4. *The translation enc_2 is adequate.*

Proof. Since types are removed and the encoding translates every syntactic construct separately, enc_2 is compositional. Hence, by Proposition 2.1, it suffices to show convergence equivalence of enc_2, i.e. we have to show four parts:

1. $p\downarrow \Rightarrow enc_2(p)\downarrow$. This follows by induction on the length of a successful reduction for p. For the base case, Lemma B.2 shows that $enc_2(p)$ is successful when p is. The induction step follows by the simulation Lemma B.3.
2. $enc_2(p)\downarrow \Rightarrow p\downarrow$. We use induction on the length of a successful reduction $enc_2(p) \xrightarrow{\mathsf{ev},k} p_0$. The base case obviously holds. For the induction step let $enc_2(p) \xrightarrow{\mathsf{ev}} p' \xrightarrow{\mathsf{ev},k-1} p''$ where p'' is successful.
   We first argue that p cannot be an irreducible non-value. Due to typing of p, then p would be an open process of the form $D[E[(x\ e)]]$, $D[E[\mathbf{case}\ x\ \ldots]]$, $D[E[\mathbf{exch}(x,v)]$ or $D[F[x]]$ where x is a free variable. For all cases it is easy to verify that $enc_2(p)$ would be an irreducible value non-value, contradicting $enc_2(p)\downarrow$.
   If the redex is not the translation of an CASE.BETA(ev)-redex, then the reduction can be performed directly for p, i.e. $p \xrightarrow{\mathsf{ev}} p_1$ with $p' = enc_2(p_1)$ holds. The induction hypothesis implies $p_1\downarrow$ and thus $p\downarrow$. If the reduction $enc_2(p) \xrightarrow{\mathsf{ev}} p'$ is the beginning of an encoded CASE.BETA(ev), then we perform the complete encoded CASE.BETA(ev)-reduction using β-CBV(ev)-reductions for $\lambda(\mathsf{f})$: $enc_2(p) \xrightarrow{\mathrm{BETA}(\mathsf{ev}),*} p_2$, $p \xrightarrow{\mathrm{CASE.BETA}(\mathsf{ev})} p_2'$, and $p_2 = enc_2(p_2')$. An illustration of the situation is shown on the right.



   Now one needs to verify that the following holds in $\lambda(\mathsf{f})$: If $q \xrightarrow{\mathsf{ev},k} q'$ where q' is successful and $q \xrightarrow{\mathrm{BETA}(\mathsf{ev})} q''$ then $q'' \xrightarrow{\mathsf{ev},\leq k} q'''$ where q''' is successful. This is easy to prove, since BETA(ev)-reductions commute with other standard reductions, as the diagram shows.



   Using this fact we have a successful reduction for $enc(p_2')$ of length $\leq k$. From the induction hypothesis we derive $p_2'\downarrow$ and thus $p\downarrow$.
3. $p\Downarrow \Rightarrow enc_2(p)\Downarrow$. We show the equivalent claim $enc_2(p)\uparrow \Rightarrow p\uparrow$, by an induction on the length of a reduction starting with $enc_2(p)$ and ending in a must-divergent process. For the base $enc_2(p)\Uparrow$ and thus by part 1, p is must-divergent. The induction step is analogous to part 2.
4. $enc_2(p)\Downarrow \Rightarrow p\Downarrow$. The equivalent claim $p\uparrow \Rightarrow enc_2(p)\uparrow$ is proved by induction on the length of a reduction for p ending in a must-divergent process. The proof is analogous to part 1, and the base case is covered by part 2. □

Corollary B.5. *The following transformations are correct for $\lambda^{\tau}(\mathsf{fc}_{av})$:*

- *All standard reductions except for* CELL.EXCH(ev).

- *The transformations* β-CBV(a), CASE.BETA(a), FUT.DEREF(a), CELL.DEREF, GC *and* DET.EXCH *(See Fig. 7, with definitions adjusted to* $\lambda^\tau(\mathsf{fc}_{av})$*).*

Proof. Let $\approx$ be one of the mentioned transformations. Due to adequacy of enc_2 it suffices to show $\Delta \vdash enc_2(p) \sim enc_2(p')$ for every $p \approx p'$ with $\Delta \vdash p$ and $\Delta \vdash p'$. This follows from Corollary A.3 and the fact that the CASE.BETA(a)-reduction can be performed as a sequence of β-CBV(a) reductions in the image of the encoding. □

B.2 Adequacy of enc_1

Let $\overline{enc_1}$ be the backtranslation from labelled $\lambda^\tau(\mathsf{fc}_{av})$-processes into $\lambda^\tau(\mathsf{fc})$-processes, which is homomorphic except for the case of labelled λ abstractions:

$$\overline{enc_1}((\lambda^{l_1} x_1.\ \ldots \lambda^{l_n} x_n.e)\ e_1^{l_1}\ \ldots\ e_n^{l_n}) \triangleq \overline{enc_1}(e)[\overline{enc_1}(e_1)/x_1, \ldots \overline{enc_1}(e_n)/x_n]$$

Definition B.6. *Let* $p \in \lambda^\tau(\mathsf{fc}_{av})$ *be a labelled process. We say that the labelling of* p *is* valid *iff the following conditions hold:*

- *if there is a labelled subexpression* e *with label* l_i*, then* e *is an argument of a nested application, which is of the form* $((\lambda^{l_j} x_j.\ \ldots\ \lambda^{l_i} x_i.e')\ e_j^{l_j}\ \ldots\ e_i{}^{l_i})$
- *if there is a labelled* λ *labelled with* l_i*, then the abstraction is of the following form* $\lambda^{l_j} \ldots . \lambda^{l_i} x_i.\ \ldots\ \lambda^{l_n}.k(e_1, \ldots, e_{j-1}, x_j, \ldots, x_n)$*. Moreover there are* $n - j + 1$ *arguments of a nested application and the arguments are labelled with* $j, j+1, \ldots, n$*.*

Let $p \in \lambda^\tau(\mathsf{fc}_{av})$ *be validly labelled. For a reduction of* p *the labelling is inherited as in labelled reduction, with the following exceptions:*

- *If a labelled* λ *of* p *is reduced using* β-CBV(ev)*, then the label of the lambda and the label of the argument are removed before the reduction is performed.*
- *If an expression is copied using* FUT.DEREF(ev)*,* CASE.BETA(ev)*, or* β-CBV(ev) *and the copied expression contains labels, then the labels are also copied but renamed with fresh labels.*

Obviously, for every process $p \in \lambda^\tau(\mathsf{fc})$ its encoding $enc_1(p)$ is validly labelled.

Lemma B.7. *For all* $p \in \lambda^\tau(\mathsf{fc})$*:* $\overline{enc_1}(enc_1(p)) = p$ *and for all validly labelled* $p \in \lambda^\tau(\mathsf{fc}_{av})$*:* $enc_1(\overline{enc_1}(p)) \xrightarrow{\beta\text{-CBV}(\mathsf{a}),*} p$

Proof. The first part is obvious from the definitions of $\overline{enc_1}$ and enc_1. The second part is by induction on p. The exceptional case is an unlabelled constructor application $k(v_1, \ldots, v_n)$ so that $\overline{enc_1}$ yields $k(\overline{enc_1}(v_1), \ldots, \overline{enc_1}(v_n))$, but enc_1 abstracts the constructor application resulting in $(\lambda x_1, \ldots, x_n.k(x_1, \ldots, x_n))\ enc_1(\overline{enc_1}(v_1))\ \ldots\ enc_1(\overline{enc_1}(v_n))$. Nevertheless, β-CBV(a)-reductions can be applied, since by induction hypothesis, every $enc_1(\overline{enc_1}(v_i))$ can be reduced to v_i. The result of this is $(\lambda x_1, \ldots, x_n.k(x_1, \ldots, x_n))\ v_1\ \ldots\ v_n$. Since each v_i is a $\lambda^\tau(\mathsf{fc}_{av})$-value, β-CBV(a) can be used to to reduce this further to $k(v_1, \ldots, v_n)$. □

Lemma B.8. *For all processes of $\lambda^\tau(\mathsf{fc}_{av})$: $p \xrightarrow{\mathsf{ev}} p' \Rightarrow p' \in \lambda^\tau(\mathsf{fc}_{av})$. Moreover, if p is validly labelled, then p' is validly labelled.*

Proof. By inspecting the reduction rules of $\lambda^\tau(\mathsf{fc})$, and the inheritance rules for the labelling. □

Let p be a validly labelled process of $\lambda^\tau(\mathsf{fc}_{av})$ with $p \xrightarrow{\mathsf{ev}} q$. We call this reduction a-labelled if and only if it is by a BETA(ev)-reduction of a labelled abstraction. If $p \xrightarrow{\mathsf{ev}} q$ and the (inner) redex is inside a labelled expression, and the reduction is not already a-labelled, then we say the reduction is b-labelled. In all other cases, the reduction is called unlabelled.

Lemma B.9. *Let p be a validly labelled process, and $p \xrightarrow{\mathsf{ev}} p'$.*

- *If the reduction is a-labelled, then $\overline{enc_1}(p) = \overline{enc_1}(p')$.*
- *If the reduction is b-labelled or unlabelled, then $\overline{enc_1}(p) \xrightarrow{\mathsf{ev}} \overline{enc_1}(p')$.*

Proof. The first part follows since the a-labelled reduction performed for p is also performed when calculating $\overline{enc_1}(p)$. For the second part we distinguish two cases: If the reduction is b-labelled, then the position of the reduction for p is moved to another position in $\overline{enc_1}(p)$. Nevertheless, since the labelling of p is valid, this reduction can be performed as a standard reduction. If the reduction is unlabeled, then it is easy to verify that the reduction can be performed for $\overline{enc_1}(p)$, since $\overline{enc_1}$ does not change the corresponding position. □

Lemma B.10. *Let $p_0 \in \lambda^\tau(\mathsf{fc}_{av})$ be a validly labelled process with $p_0\!\downarrow$, i.e. $p_0 \xrightarrow{\mathsf{ev},k} p_k$, p_k successful. Then $\overline{enc_1}(p_0)\!\downarrow$.*

Proof. We use induction on k. The base case is obvious. For the induction step let $p_0 \xrightarrow{\mathsf{ev}} p_1 \xrightarrow{\mathsf{ev},k-1} p_k$. From the induction hypothesis we have $\overline{enc_1}(p_1)\!\downarrow$. For the reduction $p_0 \xrightarrow{\mathsf{ev}} p_1$: If the reduction is a-labelled, then $\overline{enc_1}(p_1) = \overline{enc_1}(p_0)$ and the claim holds. Otherwise, the reduction is b-labelled or unlabeled. Then Lemma B.9 shows that $\overline{enc_1}(p_0) \xrightarrow{\mathsf{ev}} \overline{enc_1}(p_1)$. Hence, the claim holds. □

Corollary B.11. *For all p, $enc_1(p)\!\downarrow \Rightarrow p\!\downarrow$.*

Proof. Assume $enc_1(p)\!\downarrow$. Lemma B.10 shows $\overline{enc_1}(enc_1(p))\!\downarrow$, and thus $p\!\downarrow$. □

Lemma B.12. *Let v be a $\lambda^\tau(\mathsf{fc})$-value. Then there exists a $\lambda^\tau(\mathsf{fc}_{av})$-value w such that for all process contexts D and expression contexts C, there is a transformation $D[C[enc_1(v)]] \xrightarrow{\beta\text{-CBV}(\mathsf{a}),*} D[C[w]]$.*

Proof. This follows by structural induction on the value v. The only exceptional case are constructor applications, when some a-labelled reductions are necessary to obtain values from the translation of $enc_1(k(\vec{v}))$. □

Lemma B.13. *For the translation enc_1 the following holds:*

- *If D is a process context of $\lambda^\tau(\mathsf{fc})$, then $enc(D)$ is a process context.*

- *For every evaluation context* $\widetilde{E}$ *of* $\lambda^{\tau}(\mathsf{fc})$ *there exists an evaluation context* $\widetilde{E}'$ *such that for all expressions* e*, process contexts* D*, expression contexts* C*, and variables* x *such that* $D[x{\Leftarrow}C[enc_1(\widetilde{E})[e]]]$ *is well-typed, we have the transformation* $D[x{\Leftarrow}C[enc_1(\widetilde{E})[e]]] \xrightarrow{\beta\text{-CBV}(\mathsf{a}),*} D[x{\Leftarrow}C[\widetilde{E}'[e]]]$.
- *For every future evaluation context* $\widetilde{F}$ *of* $\lambda^{\tau}(\mathsf{fc})$*, there exists an future evaluation context* $\widetilde{F}'$ *such that for all expressions* e *and process contexts* D *and expression contexts* C *and variables* x *such that* $D[x{\Leftarrow}C[enc_1(\widetilde{F})[e]]]$ *is well-typed we have* $D[x{\Leftarrow}C[enc_1(\widetilde{F})[e]]] \xrightarrow{\beta\text{-CBV}(\mathsf{a}),*} D[x{\Leftarrow}C[\widetilde{F}'[e]]]$.

Proof. By inspecting all cases for process, evaluation, and future evaluation contexts, and using Lemma B.12 to obtain values from translated values. □

Lemma B.14. *For all* $p_1, p_2 \in \lambda^{\tau}(\mathsf{fc})$ *with* $p_1 \xrightarrow{\mathsf{ev}} p_2$ *we have* $enc_1(p_1) \xrightarrow{\beta\text{-CBV}(\mathsf{a}),*} p' \xrightarrow{\mathsf{ev}} p'' \xleftarrow{\beta\text{-CBV}(\mathsf{a}),*} enc_1(p_2)$.

Proof. By Lemma B.13, and inspection of all standard reduction rules. □

Lemma B.15. *For all processes* p *of* $\lambda^{\tau}(\mathsf{fc})$*,* $p{\downarrow} \Rightarrow enc_1(p){\downarrow}$.

Proof. This follows by induction on the length of a successful reduction of p. For the base case, let p be a successful process. Then we can transform $enc_1(p)$ into a successful process using Lemma B.12. These transformations are correct by Corollary B.5, and thus $enc_1(p){\downarrow}$. For the induction step let $p \xrightarrow{\mathsf{ev}} p'$ be the first reduction for p. By the induction hypothesis we have $enc_1(p'){\downarrow}$. We now apply Lemma B.14 to $p \xrightarrow{\mathsf{ev}} p'$, i.e., $enc_1(p) \xrightarrow{\beta\text{-CBV}(\mathsf{a}),*} p_1 \xrightarrow{\mathsf{ev}} p_2 \xleftarrow{\beta\text{-CBV}(\mathsf{a}),*} enc(p')$. Since β-CBV(a) is correct (Corollary B.5) we have $p_2{\downarrow}$ and thus $p_1{\downarrow}$. Applying correctness of β-CBV(a) again yields $enc_1(p){\downarrow}$. □

Lemma B.16. *For all validly labelled processes* p *of* $\lambda^{\tau}(\mathsf{fc}_{av})$*,* $p{\uparrow} \Rightarrow \overline{enc_1}(p){\uparrow}$.

Proof. The proof is by induction on the length of a successful reduction for p. For the base case we show $p{\Uparrow} \Rightarrow \overline{enc_1}(p){\Uparrow}$ by the equivalent claim $\overline{enc_1}(p){\downarrow} \Rightarrow p{\downarrow}$. So let us assume $\overline{enc_1}(p){\downarrow}$. Then by Lemma B.15 $enc_1(\overline{enc_1}(p)){\downarrow}$. From Lemma B.7 we have $enc_1(\overline{enc_1}(p)) \xrightarrow{\beta\text{-CBV}(\mathsf{a}),*} p$. Correctness of β-CBV(a) (Corollary B.5) shows the required $p{\downarrow}$. The induction step is analogous to Lemma B.10. □

Corollary B.17. *For all* p*,* $p{\Downarrow} \Rightarrow enc_1(p){\Downarrow}$.

Proof. The claim is equivalent to $enc_1(p){\uparrow} \Rightarrow p{\uparrow}$, which follows from Lemma B.16 since $\overline{enc_1}(enc_1(p)) = p$ by Lemma B.7. □

Proposition B.18. *The translation* enc_1 *is adequate.*

Proof. enc_1 is obviously compositional. Hence, it remains to show convergence equivalence. We have to show that may- and must-convergence are preserved and reflected by enc_1. From Corollary B.11, Lemma B.15, and Corollary B.17 we obtain three of the four parts. Hence, it remains to show that $enc_1(p){\Downarrow} \Rightarrow p{\Downarrow}$

holds. The equivalent claim $p\!\uparrow \Rightarrow enc_1(p)\!\uparrow$ can be proved by induction on the length of a reduction sequence for p ending in a must-divergent expression. The base case is covered by Corollary B.11, and the induction step is analogous to Lemma B.15. □

By combining the Propositions B.4 and B.18, and using that composition preserves adequacy, we obtain the following theorem:

Theorem B.19. *The translation enc is adequate.*

It remains to show correctness of program transformations for $\lambda^{\tau}(\mathsf{fc})$.

Corollary B.20. *The following program transformations are correct in* $\lambda^{\tau}(\mathsf{fc})$*:*

- *All reduction rules of* $\lambda^{\tau}(\mathsf{fc})$*, except for* CELL.EXCH(ev).
- *The transformations* β-CBV(a), FUT.DEREF(a), CELL.DEREF, GC, DET.EXCH, *and* CASE.BETA(a) *(see Fig. 7).*

Proof. We must show that $\Delta \vdash p \sim p'$ for every $p \approx p'$ such that $\Delta \vdash p$ and $\Delta \vdash p'$, with $\approx$ any of the listed transformations. By adequacy and the fact that enc_1 acts as identity on types, it suffices to prove $\Delta \vdash enc_1(p) \sim enc_1(p')$. But this is guaranteed by the correctness results for $\lambda^{\tau}(\mathsf{fc}_{av})$ (Corollary B.5) and using Lemma B.13 to obtain values, Lemma B.12 to obtain process, evaluation and future evaluation contexts in the image of enc_1. □

C On the Translation T_B from $\lambda^{\tau}(\mathsf{fcb})$ to $\lambda^{\tau}(\mathsf{fc})$

Let $Ctxt(\Delta';\Delta)$ denote the contexts C such that $\Delta' \vdash p \Rightarrow \Delta \vdash C[p]$, and let $Ctxt(\Gamma,\tau;\Delta)$ be the set of contexts C such that $\Gamma \vdash e:\tau \Rightarrow \Delta \vdash C[e]$.

Lemma C.1 (Compositionality). *The translation* $T_B : \lambda^{\tau}(\mathsf{fcb}) \to \lambda^{\tau}(\mathsf{fc})$ *is compositional, i.e., for all* e,p,C,D *we have* $T_B(C)[T_B(e)] = T_B(C[e])$ *and* $T_B(D)[T_B(p)] = T_B(D[p])$.

Proof. Immediate from the fact that T_B is extended homomorphically from constants to all terms, and from base components to arbitrary processes, resp. □

Lemma C.2 (Type correctness). *Let* e *and* p *be* $\lambda^{\tau}(\mathsf{fcb})$*-terms and processes, and* C*,*D *be* $\lambda^{\tau}(\mathsf{fcb})$*-term and -process contexts, respectively.*

1. *If* $\Gamma \vdash e:\tau$ *then* $T_B(\Gamma) \vdash T_B(e) : T_B(\tau)$.
2. *If* $\Delta \vdash p$ *then* $T_B(\Delta) \vdash T_B(p)$.
3. *If* $C \in Ctxt(\Gamma,\tau;\Delta)$ *then* $T_B(C) \in Ctxt(T_B(\Gamma),T_B(\tau);T_B(\Delta))$.
4. *If* $D \in Ctxt(\Delta_1;\Delta_2)$ *then* $T_B(D) \in Ctxt(T_B(\Delta_1);T_B(\Delta_2))$.

Proof.

1. The proof is by induction on the typing derivation. The main work is in the case of the buffer constants, where one checks that the implementations given in Figure 12 can be typed in $\lambda^{\tau}(\mathsf{fc})$ with all the instances of the (translations of) type schemes from Figure 11.

2. Again by induction on the typing derivation. Analogously to the above, one shows that the translation of processes $x\,\mathsf{b}-$ and $x\,\mathsf{b}\,v$ have type $T_B(\Gamma; x{:}T_B(\mathsf{buf}\ \tau)) \vdash \ldots$ whenever $x \notin dom(\Gamma)$. For latter we can additionally assume $\Gamma, x{:}\mathsf{buf}\ \tau \vdash v : \tau$, and thus $T_B(\Gamma), x{:}T_B(\mathsf{buf}\ \tau) \vdash T_B(v) : T_B(\tau)$ by the above. The remaining cases then follow immediately by induction hypothesis.
3. We first prove for *term-valued* contexts C that whenever there exists $T_B\Gamma \vdash e : T_B\tau$ such that $T_B\Gamma' \not\vdash C[e] : T_B\tau'$, then there also exists some e' such that $\Gamma \vdash e' : \tau$ and $T_B\Gamma' \not\vdash C[T_B(e')] : T_B\tau'$. In other words, witnesses for a type failure can be found in the image of the translation. A corresponding property then holds for $\lambda^\tau(\mathsf{fcb})$-term contexts.
   Now assuming that there exists $T_B\Gamma \vdash e : T_B\tau$ but $T_B\Delta \not\vdash (T_BC)[e]$, then this gives us e such that $\Gamma \vdash e : \tau$ and $T_B\Delta \not\vdash (T_BC)[T_B(e)]$. By compositionality, the latter is expressed equivalently as $T_B\Delta \not\vdash T_B(C[e])$. Hence by part (2), $\Delta \not\vdash C[e]$. But since $\Gamma \vdash e : \tau$ it cannot be the case that $C \in \mathit{Ctxt}(\Gamma, \tau; \Delta)$. We therefore have proved that $C \in \mathit{Ctxt}(\Gamma, \tau; \Delta)$ implies $T_B(C) \in \mathit{Ctxt}(T_B(\Gamma), T_B(\tau); T_B(\Delta))$.
4. Similarly to the previous case. □

C.1 Invariants of the Buffer-Implementation

In this subsection we analyze the global state of the $\lambda^\tau(\mathsf{fc})$-translation $T_B(p)$ during reduction. The argument will be used for any number of buffers, but we will restrict the arguments mainly to the situation where one buffer is used.

We describe the state of $T_B(p)$ during evaluation, where we only focus on the part that implements the buffer x. We also describe in detail the active instances of *put* and *get*, i.e., the code-pointer and internal state of the implemented *put*s and *get*s until they are finished. The notation in general is $f_{x,g,i}, h_{x,g,i}, f_{x,p,i}, h_{x,p,i}$ for futures and handles and $n_{x,p}, n_{x,g}$ for the number of processes for the buffer x, and g for *get*, p for *put*, and i is an index. In the following we omit the buffer x in the notation if it is not ambiguous.

- For the active get-functions: For $i = 0, \ldots, n_g$ there are futures $f_{g,i}$ and the corresponding handles $h_{g,i}$, and also the futures $f'_{g,i}$ and the corresponding handles $h'_{g,i}$,
- For the active *put*-functions: For $i = 0, \ldots, n_p$, there are futures $f_{p,i}$ and the corresponding handles $h_{p,i}$.
- For active *put*- and *get*-functions, we index the corresponding code-pointer in the functions with $i = 1, \ldots, n_p$ and $i = 1, \ldots, n_g$, and the code-pointer may have values 1a, 1b, 2, 3a, 3b as in the encoding in Fig. 12, where the indexing is according to the sequential execution. For the **put**-encoding:

  (1) wait (**exch**(x_p, f)); (1a) for **exch** …; (1b) for wait
  (2) **exch**(x_s, v_i);
  (3) (**exch**$(x_h, h_{p,i})$)(True) (3a) for **exch** …; (3b) for handle-binding

For the **get**-encoding:

(1)	wait (**exch** x_g f);	(1a) for **exch** ...; (1b) for wait
(2)	let $v = \mathbf{exch}(x_s, f'_{g,i})$;	
(3)	in (**exch**$(x_h, h_{g,i})$)(True); v	(3a) for **exch** ...; (3b) for handle-binding

We can assume that the initial generation of handle-future pairs is executed together with the evaluation of the first exchange-operation. Other reductions, like beta-reduction of the let and other beta-, case- and deref-reductions, are not explicitly mentioned in our analysis.

- With k we denote the index of the active *get* or *put* function that has currently access to the cell x_s of the buffer.
- We let $f_{p,0} = \mathsf{True}$. The variables $f_{g,0}, h_{g,0}$, $f'_{g,0}, h'_{g,0}$ are the future-handle pairs from the execution of **newBuf** that generates the buffer x (or from the initial translation of buffer-components).
- The current values (possibly futures) stored in the cells are x_g, x_p, x_s, x_h. Initially: $\langle x_p, x_g, x_s, x_h\rangle = \langle \mathsf{True}, f_{g,0}, f'_{g,0}, h_{g,0}\rangle$.
- The only synchronizations occur at the wait-command, where the execution has to wait until the handle corresponding to the future is bound to True by the corresponding handle-bind.

The current bindings of the futures $f_{g,i}$ and $f_{p,i}$ do not contribute, since they can only be bound to True, or are not yet available due to a missing handle-bind. An active *put* with index i has one of the following possible states, and can be interrupted at any of these points:

$1a$	... (**exch**(x_p, f));	$x_p := f_{p,i}$ where $n_p = i$
$1b$	wait $f_{p,i-1}$	synchronization
2	**exch**(x_s, v_i);	provided $h_{p,i}$ has been bound
		$x_s := v_i$ where now $k := i$
$3a$	(**exch**$(x_h, h_{p,i})$)(True)	$x_h := h_{p,i}$
$3b$	$(h_{g,i-1})$(True)	handle $h_{g,i-1}$ will be used: $f_{g,i-1} \Leftarrow \mathsf{True}$

The state of an active *get* with index i has one of the following possibilities, and can be interrupted at any of these points:

$1a$	... (**exch**(x_g, f));	$x_g := f_{g,i}$ where $n_g = i$;
$1b$	wait $g_{p,i-1}$	synchronization
2	let $v = \mathbf{exch}(x_s, f'_{g,i}) \ldots$	provided $h_{g,i}$ has been bound
	let $v = v_i \ldots$	$x_s := f'_{g,i}$ where now $k := i$
$3a$	$\ldots(\mathbf{exch}(x_h, h_{g,i}))\ldots$	$x_h := h_{g,i}$
$3b$	$h_{p,i}$(True)	$h_{p,i}$ will be used, $f_{p,i} \Leftarrow \mathsf{True}$ and v returned

The complete execution of *newBuf* creates four cells and returns a 4-tuple of these cells. During reduction there are several possibilities for $\langle x_p, x_g, x_s, x_h\rangle$:

$\langle f_{p,0}, f_{g,0}, f'_{g,0}, h_{g,o}\rangle$ Initially, i.e. after *newBuf*, where $k = n_p = n_g = 0$
$\langle f_{p,n_p}, f_{g,n_g}, v_k, h_{g,k-1}\rangle$ after 2 of $P_{p,k}$
$\langle f_{p,n_p}, f_{g,n_g}, v_k, h_{p,k}\rangle$ after 3a of $P_{p,k}$
$\langle f_{p,n_p}, f_{g,n_g}, f'_{g,n_g}, h_{p,k}\rangle$ after 2 of $P_{g,k}$
$\langle f_{p,n_p}, f_{g,n_g}, f'_{g,n_g}, h_{g,k}\rangle$ after 3a of $P_{g,k}$

A further invariant is that the occurrences of the handles and futures, and also of the reference cells belonging to x, are only at the mentioned places, and nowhere else. Note this is only true since we look for the images of processes by the implementation T_B.

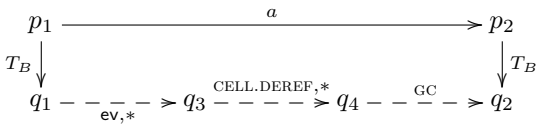
Informally, there are two queues: one of active *put*-instances waiting for activating the respective handles by the previous *get*, and a queue of *get*-instances waiting for activating the handle by the previous *put*. All the futures will be bound to True at some time, provided the evaluation terminates successfully.

Using induction on the number of buffer-related small-step reductions, we see that the above description is an invariant for the buffer-implementation of a single buffer x.

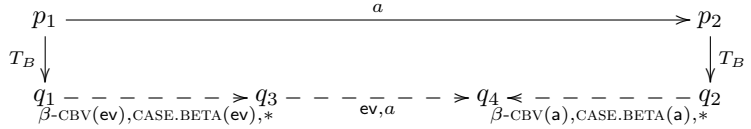
C.2 Convergence Equivalence of T_B

Proposition C.3. *For every $\lambda^\tau(\mathsf{fcb})$-process p, $p{\downarrow} \Rightarrow T_B(p){\downarrow}$.*

Proof. We use induction on the number of reduction steps in a fixed reduction U corresponding to $p{\downarrow}$ in order to construct a transformation sequence U_c of $T_B(p)$ to a successful process, where Lemma 4.1 shows the base case. The translation T_B modifies the evaluation contexts and future evaluation contexts, but only in a predictable way. We use the following diagrams to construct for every reduction step a corresponding sequence of transformations and reductions. We omit the trivial diagrams where a reduction is simply mirrored. The following diagrams can be completed:



where $a \in \{\text{BUFF.PUT}(\mathsf{ev}), \text{BUFF.GET}(\mathsf{ev}), \text{BUFF.NEW}(\mathsf{ev})\}$; and the corresponding reduction $q_1 \xrightarrow{*} q_3$ is the complete reduction of the respective body of the translation including the dereferences before the wait followed by perhaps the transformation CELL.DEREF, then followed by the transformation GC that removes certain no longer used handle- and thread-components.



where $a \in \{\text{FUT.DEREF}(\mathsf{ev}), \text{LAZY.TRIGGER}(\mathsf{ev})\}$; and the first sequence of reductions are beta- and case-reductions of the start of the body of the *put*- or *get*-instance in order to bring the focussed variable in an $\widetilde{F}$-context, which is provided by a nesting of cases. The second sequence are the beta- and case-reductions backwards, such that $q_1 \xrightarrow{a} q_2$, however as a transformation (i.e. non-reduction).

We illustrate the diagram for a BUFF.PUT(ev)-reduction:

$$y{\Leftarrow}\mathbf{put}(x,v) \mid x\,\mathsf{b}-\longleftarrow y{\Leftarrow}\mathsf{unit} \mid x\,\mathsf{b}\,v.$$

The reduction after the translation is as follows:
$y{\Leftarrow}put\langle x, T_B(v)\rangle) \mid (\nu x_p)(\nu x_g)(\nu x_s)(\nu x_h)(x{\Leftarrow}\langle x_p, x_g, x_s, x_h\rangle$
$\mid\ (\nu h)(\nu f)(\nu h')(\nu f')(h\,\mathsf{h}\,f \mid h'\,\mathsf{h}\,f' \mid x_p\,\mathsf{c}\,\mathsf{True} \mid x_g\,\mathsf{c}\,f \mid x_s\,\mathsf{c}\,f' \mid x_h\,\mathsf{c}\,h))$
will reduce to (we show also some intermediate states)
$y{\Leftarrow}\mathsf{wait}\ \mathsf{True}; \ldots \mid (\nu h_1)(\nu f_1)h_1\,\mathsf{h}\,f_1 \mid$
$(\nu x_p)(\nu x_g)(\nu x_s)(\nu x_h)(x{\Leftarrow}\langle x_p, x_g, x_s, x_h\rangle$
$\mid\ (\nu h)(\nu f)(\nu h')(\nu f')(h\,\mathsf{h}\,f \mid h'\,\mathsf{h}\,f' \mid x_p\,\mathsf{c}\,f_1 \mid x_g\,\mathsf{c}\,f \mid x_s\,\mathsf{c}\,f' \mid x_h\,\mathsf{c}\,h))$
$\rightarrow \quad y{\Leftarrow}\mathbf{exch}(x_h, h_1)\ldots \mid (\nu h_1)(\nu f_1)h_1\,\mathsf{h}\,f_1 \mid$
$(\nu x_p)(\nu x_g)(\nu x_s)(\nu x_h)(x{\Leftarrow}\langle x_p, x_g, x_s, x_h\rangle$
$\mid\ (\nu h)(\nu f)(\nu h')(\nu f')(h\,\mathsf{h}\,f \mid h'\,\mathsf{h}\,f' \mid x_p\,\mathsf{c}\,f_1 \mid x_g\,\mathsf{c}\,f \mid x_s\,\mathsf{c}\,T_B(v) \mid x_h\,\mathsf{c}\,h))$
$\rightarrow \quad y{\Leftarrow}\mathsf{unit} \mid (\nu h_1)(\nu f_1)h_1\,\mathsf{h}\,f_1 \mid$
$(\nu x_p)(\nu x_g)(\nu x_s)(\nu x_h)(x{\Leftarrow}\langle x_p, x_g, x_s, x_h\rangle$
$\mid\ (\nu h)(\nu f)(\nu h')(\nu f')(h\,\mathsf{h}\,\bullet \mid h'\,\mathsf{h}\,f' \mid x_p\,\mathsf{c}\,f_1 \mid x_g\,\mathsf{c}\,f \mid x_s\,\mathsf{c}\,T_B(v) \mid x_h\,\mathsf{c}\,h_1) \mid f{\Leftarrow}\mathsf{True})$
$\xrightarrow{\text{CELL.DEREF}} \quad y{\Leftarrow}\mathsf{unit} \mid (\nu h_1)(\nu f_1)h_1\,\mathsf{h}\,f_1 \mid$
$(\nu x_p)(\nu x_g)(\nu x_s)(\nu x_h)(x{\Leftarrow}\langle x_p, x_g, x_s, x_h\rangle$
$\quad \mid\ (\nu h)(\nu f)(\nu h')(\nu f')(h\,\mathsf{h}\,\bullet \mid h'\,\mathsf{h}\,f' \mid x_p\,\mathsf{c}\,f_1 \mid x_g\,\mathsf{c}\,\mathsf{True}$
$\quad \mid x_s\,\mathsf{c}\,T_B(v) \mid x_h\,\mathsf{c}\,h_1) \mid f{\Leftarrow}\mathsf{True})$
$\xrightarrow{\text{GC}} \quad y{\Leftarrow}\mathsf{unit} \mid (\nu h_1)(\nu f_1)h_1\,\mathsf{h}\,f_1 \mid$
$(\nu x_p)(\nu x_g)(\nu x_s)(\nu x_h)(x{\Leftarrow}\langle x_p, x_g, x_s, x_h\rangle$
$\mid\ x_p\,\mathsf{c}\,f_1 \mid x_g\,\mathsf{c}\,\mathsf{True} \mid x_s\,\mathsf{c}\,T_B(v) \mid x_h\,\mathsf{c}\,h_1)$
By induction on the length of U, the sequence U_c will be constructed such that there is a correspondence between the intermediate processes of U and U_c:

If the first reduction of U is not a BUFF.NEW(ev)-, BUFF.GET(ev)-, or BUFF.PUT(ev)-reduction, and not a FUT.DEREF(ev), LAZY.TRIGGER(ev) that is triggered through an argument-position of an BUFF.GET(ev)-, or BUFF.PUT(ev), then the same reduction is concatenated to U_c. Otherwise, we use the diagrams above to extend U_c. Thus we can assume that executing the code of *newBuf*, *get* and *put* is completely done, respectively, and not disturbed by a concurrent thread. We have constructed a transformation sequence U_c for $T_B(P)$ resulting in a successful process. However, in general this sequence is not an $\xrightarrow{\mathsf{ev}}$-reduction.

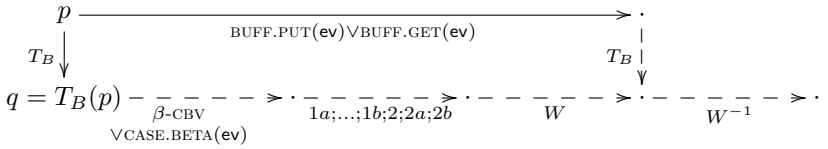
Corollary B.20 now shows that all non-ev-reductions in U_c are correct $\lambda^\tau(\mathsf{fc})$-transformations. This implies, using an induction on the length of U_c, that there is also an $\xrightarrow{\mathsf{ev}}$-reduction of $T_B(p)$ to a successful process. □

Proposition C.4. *For every $\lambda^\tau(\mathsf{fcb})$-process p, $T_B(p)\!\downarrow \Rightarrow p\!\downarrow$.*

Proof. Suppose that for the $\lambda^{\tau}(\mathsf{fcb})$-process p there is a reduction U of $T_B(p)$ to a successful process p_ω. Then the $\lambda^{\tau}(\mathsf{fcb})$-reduction of p is constructed in several steps. In the following we do not mention the beta-reductions that are used for the reduction of the let and the sequential operator ";". We will make use of the implied sequence of reductions as stated in Lemma 4.4.

1. Rearranging U: In general the single reductions of a *put*, or of a *get* are not in a single block, but may be interleaved with other reductions. We show that it is possible to gather the 1,2,3-reductions in one block:
   The idea is to commute reductions.
   (a) The $\xrightarrow{x,1b}$-reductions are moved to the right. The strategy is to start with the rightmost type-2-reduction and to move the closest type-1b-reduction to the right. I.e. a single operation is:
   $\xrightarrow{x,1b} \cdot \xrightarrow{*} \cdot \xrightarrow{x,2} \rightsquigarrow \xrightarrow{*} \cdot \xrightarrow{x,1b} \cdot \xrightarrow{x,2}$. This is possible, since all other reduction are triggered from parallel processes, and since these reductions commute with all other reductions. The result will be a rearranged reduction, where all 1b and 2-reductions of the same process are neighbors. In a similar way we move the perhaps necessary dereferencing reductions that are a precondition for the execution of the 1b-reduction (i.e. the wait) immediately before the 1b-reduction.
   (b) The same can be done for the type 1a-reductions, which will result in a reduction, where 1a, 1b and 2-reductions of the same instance are executed without other intermediate reductions. For the 1a-type it is important to note that the sequence of $\xrightarrow{x,1a}$-reductions is the same as the type-2-reductions, if the sorting criteria is the body of the *put*, *get*s.
   (c) It is also easy to see that the $\mathsf{newhandled}$-reductions can also be moved close to their corresponding 1a, 1b and 2-block.
   (d) The type 3-reductions, i.e. 3a and 3b of the put- and get-body can also be moved to the left, where the strategy is now to start with the leftmost type-2-reduction. The arguments are as in the previous items, where also the sequence of the 3a,3b-reductions (w.r.t the bodies *put*, *get*) are the same as the type-2-reductions (see Lemma 4.4).
   In a similar way we can rearrange the reduction from a *newBuf*, where the reductions before the final reduction within the body are moved to the right. In summary, we obtain an ev-reduction U_1, where the reductions of every *newBuf* are without interrupt, and where the reduction of *get* and *put* are in one block, with the exception of the first reductions which are beta- and case-reductions. The reason is that the future-strict evaluation of the arguments of **put**, **get** is mirrored after the translation by the future-strict evaluation of the arguments that will start only after the first beta-reduction for *get*, or after the first beta- and case-reduction for *put*.
2. Now we modify the reduction U_1 from left to right. This is done in a similar way as in the proof of Proposition 4.3: Whenever there is an a-reduction with $a \in \{\text{FUT.DEREF}(\mathsf{ev}), \text{LAZY.TRIGGER}(\mathsf{ev})\}$ that is triggered after the first reductions of a *put*, *get*, and it is not followed by the corresponding 1a-reduction, we immediately add a transformation sequence W consisting of

inverse beta-and case-reductions after the a-reductions where W eliminates the initial (beta,case)-reductions of *put* or *get* that triggers the a-reduction. After W, we add the inverse reduction W^{-1} immediately after W.The reduction sequence can be chosen such that W^{-1} is an ev-reduction. The following correspondence diagram then holds:



The effect is to replace $\xrightarrow{a}$ by $\xrightarrow{a} \cdot W \cdot W^{-1}$. The idea is to move the reduction steps in W^{-1} further to the right to the 1a–3b-block of the reductions, to whom they belong. This is possible, if between the beta,case-reductions and the 1a–3b-block there are no more a-reductions that are triggered by the result of the beta,case-reductions. In order to make the correspondence completely analogous to the construction in the proof of Proposition 4.3, we also have to add GC and CELL.DEREF-reductions and their inverses. This gives a sequence of correct transformations and reductions U_2

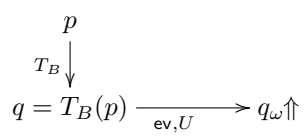
This rearrangement is performed from left to right, such that the reduction U_b of p, corresponding to U_2 via T_B, can be constructed. It is easy to see by induction that the constructed reduction is according to the ev-strategy.

Now $p \xrightarrow{U_b} p_\omega$, $T_B(p) \xrightarrow{U_2} Q \sim T_B(p_\omega)$, where $Q \xrightarrow{\text{FUT.DEREF(ev),LAZY.TRIGGER(ev),*}} T_B(p_\omega)$ and hence by Corollary B.20 and Lemma 4.1, we have $p{\downarrow}$. □

The hard part is the proof of the following proposition:

Proposition C.5. *For every $\lambda^\tau(\mathsf{fcb})$-process p: $p{\Downarrow} \Longrightarrow T_B(p){\Downarrow}$.*

Proof. We prove the equivalent claim that for every $\lambda^\tau(\mathsf{fcb})$-process p: $T_B(p){\uparrow} \Longrightarrow p{\uparrow}$.
Let p be a $\lambda^\tau(\mathsf{fcb})$-process such that $q := T_B(p){\uparrow}$. Let $U = (T_B(p) \xrightarrow{*} q_\omega)$ be a $\lambda^\tau(\mathsf{fc})$-reduction sequence with $q_\omega{\Uparrow}$.



According to the previous analysis of the intermediate states of *newBuf*, *put*, *get*-operations, and using a similar construction as in the proof of Proposition C.4, we look for all translated buffer-operations in U, for all buffers. The goal of the modification is to rearrange the reduction steps such that the reduction steps of every *newBuf*, *get*, *put* are a contiguous block of reductions, up to

the initial beta- and case-reductions. However, since the reduction is ending with a must-divergent process, there may be started executions of *newBuf*, *get*, *put* that are not finished within the reduction. So, for the proof it may be necessary to insert reduction steps. However, in order to have an induction measure, we never add reductions of type 1a, thus the number of these will not be increased. The other parts of the induction measure are then standard.

1. First we argue that the *newBuf*-reductions can be adjusted: It is easy to see that for the reductions of a single *newBuf*-instance, we can shift all the corresponding reduction steps to the right (they commute with all other reduction steps), immediately before the last reduction step of the *newBuf*, provided all of them are performed in U'. If some are missing, it is possible to perform the reductions starting from q_ω, which leads to a must-divergent process, since these are reductions according to the ev-strategy. Then we can perform the rearrangement as before. In order to have an induction measure, we do not insert the first beta-reduction of an *newBuf*.
2. Now we rearrange the reduction steps of *put*, *get*, where we make implicit use of Lemma 4.4. Since the CELL.EXCH(ev)-reductions are not correct in general, we have to be a bit careful. First we consider the case where the $\xrightarrow{x,type-2}$ reduction is in U.
   (a) The reduction sequence $\xrightarrow{x,1b} \cdot W \cdot \xrightarrow{x,2}$, where the 1b and 2 reduction step are activated by the same body, is rearranged as follows: $W \cdot \xrightarrow{x,1b} \cdot \xrightarrow{x,2}$. This can be done in all cases.
   (b) The reduction sequence $\xrightarrow{x,1a} \cdot W \cdot \xrightarrow{x,1b} \cdot \xrightarrow{x,2}$, where the 1a and 2 reduction step are activated by the same body, is rearranged as follows: The (x,1a)-reductions in W have to be moved to the right of the sequence $W \cdot \xrightarrow{x,1b} \cdot \xrightarrow{x,2}$, keeping their sequence. Then the $\xrightarrow{x,1a}$ can be moved immediately to the left of $\xrightarrow{x,2}$. Lemma 4.4 justifies these moves, and since the other commutations are possible. We perform this exhaustively.
   (c) Now we shift the (x,3a) and (x,3b)-reductions to the left. This can be done using the strategy by starting with the rightmost (x,3a)-reduction, then the rightmost (x,3b)-reduction, and so on. However, there may be (x,2)-reductions without corresponding (x,3a),(x,3b)-reduction in the sequence. In this case, the first missing (x,3a)-reduction is also a ev-reduction for q_ω, thus the result is again must-divergent, and so we can insert it after q_ω, i.e. assume that it is already in the reduction. The same holds for for the first missing (x,3b)-reduction.
3. Now we consider the case where some type (x,1a)-reduction is in the sequence, but there is no corresponding continuation. Let us consider the leftmost (x,1a)-reduction with missing (x,1b)-reduction. As above, this is an ev-reduction for q_ω and hence we can assume that it is already in the reduction sequence. The same for (x,1b)- and (x,2)-reductions. Note that between (x,1a) and (x,1b) there may be some dereferencing, which can be shifted along with the other reductions and can be treated in a standard way.
   If a process has performed the newhandled-reduction(s), but the (x,1a)-reduction is not in the sequence, then this can be shifted in the reduction

such that it ends with $q' \xrightarrow{\mathsf{newhandled}} q_\omega$. Then Corollary B.20 shows that q' is also must-divergent, and we can remove the newhandled-operation.

4. Treatment of triggering dereferencing and lazy-trigger can done be as usual.

Now the situation is as follows: There is some process p_ω with $p \xrightarrow{*} p_\omega$ and $T_B(p_\omega) = q_\omega$ and $q_\omega\Uparrow$. Proposition 4.3 shows that $p_\omega\downarrow$ is impossible, hence $p_\omega\Uparrow$, which implies $p\uparrow$.

C.3 Lifting Equivalences to $\lambda^\tau(\mathsf{fcb})$

Lemma C.6. *The following equivalences hold in* $\lambda^\tau(\mathsf{fcb})$*:*

- *All reduction rules of* $\lambda^\tau(\mathsf{fcb})$ *except for* CELL.EXCH(ev) *and* BUFF.PUT(ev), BUFF.GET(ev) *are correct.*
- *The transformations* β-CBV(a), FUT.DEREF(a), CELL.DEREF, GC *and* DET.EXCH *(see Fig. 7) lifted to* $\lambda^\tau(\mathsf{fcb})$ *are correct.*

Proof. Let $\approx$ be a program transformation mentioned in the claim. Due to adequacy of T_B, it is sufficient to show $T_B(\Delta) \vdash T_B(p_1) \sim T_B(p_2)$ for all $(p_1, p_2) \in \approx$ with $\Delta \vdash p_i$, $i = 1, 2$. As already argued (see Lemma C.2) the translation T_B of D- and E-contexts of $\lambda^\tau(\mathsf{fcb})$ results in D and E-contexts of $\lambda^\tau(\mathsf{fc})$. The same holds for values. Thus the correctness of the reduction rules β-CBV(ev), THREAD.NEW(ev), HANDLE.NEW(ev), HANDLE.BIND(ev), CELL.NEW(ev), LAZY.NEW(ev) and CASE.BETA(ev) is easy to obtain by using the according equivalence for $\lambda^\tau(\mathsf{fc})$. For the rules using F-contexts (FUT.DEREF(ev) and LAZY.TRIGGER(ev)), similar as in the proof of Proposition C.3 we can show that the equivalence holds: A reduction $\xrightarrow{a}$ with ($a \in \{$FUT.DEREF(ev), LAZY.TRIGGER(ev)$\}$ corresponds by T_B to a transformation of the form $\xrightarrow{\beta\text{-CBV(ev)},\text{CASE.BETA(ev)},*} \cdot \xrightarrow{a} \cdot \xleftarrow{\beta\text{-CBV(ev)},\text{CASE.BETA(ev)},*}$, which is a correct transformation. The correctness of β-CBV(a), FUT.DEREF(a), GC, DET.EXCH and CELL.DEREF follows directly by using the equivalences for the encodings in $\lambda^\tau(\mathsf{fc})$. Correctness of BUFF.NEW(ev) follows by inspecting the encoding (see proof of Proposition C.3): The translation only consists of reductions that are correct transformations of $\lambda^\tau(\mathsf{fc})$. □

C.4 Correctness of Axiomatic Laws

Finally, we prove the correctness of the axiomatic laws at the end of Section 4.

Proposition C.7. *Using Corollary B.20, we show that the following rules hold for* **put** *and* **get***:*

$$\begin{array}{ll} (\text{DET.PUT}) & (\nu x).E[\mathbf{put}(x,v)] \mid x\,\mathsf{b}\,- \;\to (\nu x).E[\mathbf{unit}] \mid x\,\mathsf{b}\,v \\ (\text{DET.GET}) & (\nu x).E[\mathbf{get}\ x] \mid x\,\mathsf{b}\,v \;\to (\nu x).E[v] \mid x\,\mathsf{b}\,- \end{array}$$

Proof. The proof is done by using the adequate encoding of buffers in $\lambda^{\tau}(\mathsf{fc})$, and the valid equivalences in $\lambda^{\tau}(\mathsf{fc})$. Using the encoding of buffers, we focus on the translation of the expression $(\nu x).E[\mathbf{put}(x,v)] \mid x\,\mathsf{b}\,-$. This results in

$$\begin{array}{c}(\nu x)(\nu x_p)(\nu x_g)(\nu x_s)(\nu x_h)(\nu h)(\nu f)(\nu h')(\nu f').\\ E[put\langle x,v\rangle] \mid x{\Leftarrow}\langle x_p,x_g,x_s,x_h\rangle\\ \mid h\,\mathsf{h}\,f \mid h'\,\mathsf{h}\,f' \mid x_p\,\mathsf{c}\,\mathsf{True} \mid x_g\,\mathsf{c}\,f \mid x_s\,\mathsf{c}\,f' \mid x_h\,\mathsf{c}\,h\end{array}$$

(Strictly speaking, the context E must be translated as well. But its translation is another EC.) Reducing *put* and applying dereferencing and case-reductions, we obtain:

$$\begin{array}{c}(\nu x)(\nu x_p)(\nu x_g)(\nu x_s)(\nu x_h)(\nu h)(\nu f)(\nu h')(\nu f')(\nu f_1)(\nu h_1)\\ E[\mathbf{put}^1[x,v]] \mid h_1\,\mathsf{h}\,f_1 \mid x{\Leftarrow}\langle x_p,x_g,x_s,x_h\rangle\\ \mid h\,\mathsf{h}\,f \mid h'\,\mathsf{h}\,f' \mid x_p\,\mathsf{c}\,\mathsf{True} \mid x_g\,\mathsf{c}\,f \mid x_s\,\mathsf{c}\,f' \mid x_h\,\mathsf{c}\,h\end{array}$$

The notation $\mathbf{put}^i[x,v]$ means $put\langle x,v\rangle$ after some reductions. Executing exchange of f_1 in x_p and reducing wait results in:

$$\begin{array}{c}(\nu x),(\nu x_p)(\nu x_g)(\nu x_s)(\nu x_h)(\nu h)(\nu f)(\nu h')(\nu f')(\nu f_1)(\nu h_1)\\ E[\mathbf{put}^2[x,v]] \mid h_1\,\mathsf{h}\,f_1 \mid x{\Leftarrow}\langle x_p,x_g,x_s,x_h\rangle\\ \mid h\,\mathsf{h}\,f \mid h'\,\mathsf{h}\,f' \mid x_p\,\mathsf{c}\,f_1 \mid x_g\,\mathsf{c}\,f \mid x_s\,\mathsf{c}\,f' \mid x_h\,\mathsf{c}\,h\end{array}$$

The next reduction is to exchange v in the cell x_s, resulting in

$$\begin{array}{c}(\nu x),(\nu x_p)(\nu x_g)(\nu x_s)(\nu x_h)(\nu h)(\nu f)(\nu h')(\nu f')(\nu f_1)(\nu h_1)\\ E[\mathbf{put}^3[x,v]] \mid h_1\,\mathsf{h}\,f_1 \mid x{\Leftarrow}\langle x_p,x_g,x_s,x_h\rangle\\ \mid h\,\mathsf{h}\,f \mid h'\,\mathsf{h}\,f' \mid x_p\,\mathsf{c}\,f_1 \mid x_g\,\mathsf{c}\,f \mid x_s\,\mathsf{c}\,v \mid x_h\,\mathsf{c}\,h\end{array}$$

The next reduction is to exchange h_1 in the cell x_h, resulting in

$$\begin{array}{c}(\nu x),(\nu x_p)(\nu x_g)(\nu x_s)(\nu x_h)(\nu h)(\nu f)(\nu h')(\nu f')(\nu f_1)(\nu h_1)\\ E[h\ \mathsf{True}] \mid h_1\,\mathsf{h}\,f_1 \mid x{\Leftarrow}\langle x_p,x_g,x_s,x_h\rangle\\ \mid h\,\mathsf{h}\,f \mid h'\,\mathsf{h}\,f' \mid x_p\,\mathsf{c}\,f_1 \mid x_g\,\mathsf{c}\,f \mid x_s\,\mathsf{c}\,v \mid x_h\,\mathsf{c}\,h_1\end{array}$$

The next reduction is a handle-bind, resulting in

$$\begin{array}{c}(\nu x),(\nu x_p)(\nu x_g)(\nu x_s)(\nu x_h)(\nu h)(\nu f)(\nu h')(\nu f')(\nu f_1)(\nu h_1)\\ E[\mathbf{unit}] \mid f{\Leftarrow}\mathsf{True} \mid h_1\,\mathsf{h}\,f_1 \mid x{\Leftarrow}\langle x_p,x_g,x_s,x_h\rangle\\ \mid h\,\mathsf{h}\,\bullet \mid h'\,\mathsf{h}\,f' \mid x_p\,\mathsf{c}\,f_1 \mid x_g\,\mathsf{c}\,f \mid x_s\,\mathsf{c}\,v \mid x_h\,\mathsf{c}\,h_1\end{array}$$

Now we can use a cell-dereferencing, giving

$$\begin{array}{c}(\nu x),(\nu x_p)(\nu x_g)(\nu x_s)(\nu x_h)(\nu h)(\nu f)(\nu h')(\nu f')(\nu f_1)(\nu h_1)\\ E[\mathbf{unit}] \mid f{\Leftarrow}\mathsf{True} \mid h_1\,\mathsf{h}\,f_1 \mid x{\Leftarrow}\langle x_p,x_g,x_s,x_h\rangle\\ \mid h\,\mathsf{h}\,\bullet \mid h'\,\mathsf{h}\,f' \mid x_p\,\mathsf{c}\,f_1 \mid x_g\,\mathsf{c}\,\mathsf{True} \mid x_s\,\mathsf{c}\,v \mid x_h\,\mathsf{c}\,h_1\end{array}$$

A final garbage collection leads to

$$\begin{array}{c}(\nu x)(\nu x_p)(\nu x_g)(\nu x_s)(\nu x_h)(\nu f_1)(\nu h_1)\\ E[\mathbf{unit}] \mid h_1\,\mathsf{h}\,f_1 \mid x{\Leftarrow}\langle x_p,x_g,x_s,x_h\rangle\\ \mid x_p\,\mathsf{c}\,f_1 \mid x_g\,\mathsf{c}\,\mathsf{True} \mid x_s\,\mathsf{c}\,v \mid x_h\,\mathsf{c}\,h_1)\end{array}$$

This is exactly the encoding of $(\nu x).E[\mathbf{unit}] \mid x \,\mathsf{b}\, v$ after some renaming.

The next check is whether the transformation (DET.GET) is satisfied: Now we focus on the translation of $(\nu x).E[\mathbf{get}\, x] \mid x \,\mathsf{b}\, v \sim E[v] \mid x \,\mathsf{b}\, -$ and its reductions:

$$\begin{array}{c}(\nu x).E[\mathbf{get}\ x] \mid (\nu x_p)(\nu x_g)(\nu x_s)(\nu x_h)x{\Leftarrow}\langle x_p, x_g, x_s, x_h\rangle \\ \mid (\nu h)(\nu f)h \,\mathsf{h}\, f \mid x_p \,\mathsf{c}\, f \mid x_g \,\mathsf{c}\, \mathsf{True} \mid x_s \,\mathsf{c}\, v \mid x_h \,\mathsf{c}\, h\end{array}$$

The two first new handle-reductions result in:

$$\begin{array}{c}(\nu x)(\nu x_p)(\nu x_g)(\nu x_s)(\nu x_h)(\nu h)(\nu f)(\nu h_1)(\nu f_1)(\nu h_2)(\nu f_2) \\ E[\mathbf{get}^{(1)}[x]] \mid h_1 \,\mathsf{h}\, f_1 \mid h_2 \,\mathsf{h}\, f_2 \mid x{\Leftarrow}\langle x_p, x_g, x_s, x_h\rangle \\ \mid h \,\mathsf{h}\, f \mid x_p \,\mathsf{c}\, f \mid x_g \,\mathsf{c}\, \mathsf{True} \mid x_s \,\mathsf{c}\, v \mid x_h \,\mathsf{c}\, h\end{array}$$

After a reduction of exchange of f_1 in x_g and a wait the result is:

$$\begin{array}{c}(\nu x)(\nu x_p)(\nu x_g)(\nu x_s)(\nu x_h)(\nu h)(\nu f)(\nu h_1)(\nu f_1)(\nu h_2)(\nu f_2) \\ E[\mathbf{get}^{(2)}[x]] \mid h_1 \,\mathsf{h}\, f_1 \mid h_2 \,\mathsf{h}\, f_2 \mid x{\Leftarrow}\langle x_p, x_g, x_s, x_h\rangle \\ \mid h \,\mathsf{h}\, f \mid x_p \,\mathsf{c}\, f \mid x_g \,\mathsf{c}\, f_1 \mid x_s \,\mathsf{c}\, v \mid x_h \,\mathsf{c}\, h\end{array}$$

After a reduction of exchange of f_2 in x_s the result is:

$$\begin{array}{c}(\nu x)(\nu x_p)(\nu x_g)(\nu x_s)(\nu x_h)(\nu h)(\nu f)(\nu h_1)(\nu f_1)(\nu h_2)(\nu f_2) \\ E[\mathbf{get}^{(3)}[x]] \mid h_1 \,\mathsf{h}\, f_1 \mid h_2 \,\mathsf{h}\, f_2 \mid x{\Leftarrow}\langle x_p, x_g, x_s, x_h\rangle \\ \mid h \,\mathsf{h}\, f \mid x_p \,\mathsf{c}\, f \mid x_g \,\mathsf{c}\, f_1 \mid x_s \,\mathsf{c}\, f_2 \mid x_h \,\mathsf{c}\, h\end{array}$$

After a reduction of exchange of h_1 in x_h the result is:

$$\begin{array}{c}(\nu x)(\nu x_p)(\nu x_g)(\nu x_s)(\nu x_h)(\nu h)(\nu f)(\nu h_1)(\nu f_1)(\nu h_2)(\nu f_2) \\ E[\mathbf{get}^{(3)}[x]] \mid h_1 \,\mathsf{h}\, f_1 \mid h_2 \,\mathsf{h}\, f_2 \mid x{\Leftarrow}\langle x_p, x_g, x_s, x_h\rangle \\ \mid h \,\mathsf{h}\, f \mid x_p \,\mathsf{c}\, f \mid x_g \,\mathsf{c}\, f_1 \mid x_s \,\mathsf{c}\, f_2 \mid x_h \,\mathsf{c}\, h_1\end{array}$$

After a handle-bind in $(h\ \mathsf{True})$ and reducing $(\mathbf{unit}; v)$:

$$\begin{array}{c}(\nu x)(\nu x_p)(\nu x_g)(\nu x_s)(\nu x_h)(\nu h)(\nu f)(\nu h_1)(\nu f_1)(\nu h_2)(\nu f_2) \\ E[v] \mid h_1 \,\mathsf{h}\, f_1 \mid h_2 \,\mathsf{h}\, f_2 \mid x{\Leftarrow}\langle x_p, x_g, x_s, x_h\rangle \\ \mid h \,\mathsf{h}\, \bullet \mid f{\Leftarrow}\mathsf{True} \mid x_p \,\mathsf{c}\, f \mid x_g \,\mathsf{c}\, f_1 \mid x_s \,\mathsf{c}\, f_2 \mid x_h \,\mathsf{c}\, h_1\end{array}$$

Now a cell-deref transformation results in:

$$\begin{array}{c}(\nu x)(\nu x_p)(\nu x_g)(\nu x_s)(\nu x_h)(\nu h)(\nu f)(\nu h_1)(\nu f_1)(\nu h_2)(\nu f_2) \\ E[v] \mid h_1 \,\mathsf{h}\, f_1 \mid h_2 \,\mathsf{h}\, f_2 \mid x{\Leftarrow}\langle x_p, x_g, x_s, x_h\rangle \\ \mid h \,\mathsf{h}\, \bullet \mid f{\Leftarrow}\mathsf{True} \mid x_p \,\mathsf{c}\, \mathsf{True} \mid x_g \,\mathsf{c}\, f_1 \mid x_s \,\mathsf{c}\, f_2 \mid x_h \,\mathsf{c}\, h_1\end{array}$$

Garbage collecting f, h results in:

$$\begin{array}{c}(\nu x)(\nu x_p)(\nu x_g)(\nu x_s)(\nu x_h)(\nu h_1)(\nu f_1)(\nu h_2)(\nu f_2) \\ E[v] \mid h_1 \,\mathsf{h}\, f_1 \mid h_2 \,\mathsf{h}\, f_2 \mid x{\Leftarrow}\langle x_p, x_g, x_s, x_h\rangle \\ \mid x_p \,\mathsf{c}\, \mathsf{True} \mid x_g \,\mathsf{c}\, f_1 \mid x_s \,\mathsf{c}\, f_2 \mid x_h \,\mathsf{c}\, h_1\end{array}$$

This is exactly the translation corresponding to an empty buffer $(\nu x).E[v] \mid x \,\mathsf{b}\, -$. Hence we have shown equivalence. □