On Correctness of Buffer Implementations in a Concurrent Lambda Calculus with Futures

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-37

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

May 12, 2009

Abstract. Motivated by the question of correctness of a specific implementation of concurrent buffers in the lambda calculus with futures underlying Alice ML, we prove that concurrent buffers and handled futures can correctly encode each other. Correctness means that our encodings preserve and reflect the observations of may- and must-convergence, and as a consequence also yields soundness of the encodings with respect to a contextually defined notion of program equivalence. While these translations encode blocking into queuing and waiting, we also describe an adequate encoding of buffers in a calculus without handles, which is more low-level and uses busy-waiting instead of blocking. Furthermore we demonstrate that our correctness concept applies to the whole compilation process from high-level to low-level concurrent languages, by translating the calculus with buffers, handled futures and data constructors into a small core language without those constructs.

1 Introduction

Modern concurrent programming languages extend sequential languages with concurrent threads and concurrency primitives for controlling their interactions. Computation within each thread is sequential. Examples for concurrency primitives are MVars (i.e. concurrent buffers) in Haskell [8], channels in Concurrent ML [19], handled futures in Alice ML [23], and joins in JoCaml [5].

Even though one might conjecture that many of these concurrency primitives can express each other, proving such results is an open challenge in the area of

programming language semantics. Encodings are often straightforward to find, or provided in libraries of existing programming languages. The difficult part is to introduce an appropriate program semantics for concurrent and higher-order programming languages, and to develop corresponding proof techniques for showing the correctness of an encoding.

Such correctness results of encodings are clearly of interest in many areas of programming languages. In language design these statements can help to determine an appropriate set of primitives for the language, while other constructs are provided as derived concepts. For the implementation of compilers, correctness of encodings can express that the compilation is semantics-preserving; they may also guarantee the admissibility of various program transformations. When reasoning about (concurrent) programs, correctness of encodings can be used as a technical device; by lifting program equivalences from lower-level to higher-level languages, one can abstract away from the concrete implementations and treat composite constructs as primitives.

In the present paper, we focus on two different but closely related questions. First, given two sets of synchronization primitives, how can we show their equivalence? Second, given an implementation of a "high-level" synchronization construct by a "lower-level" one, how can we show its correctness?

In order to address the question of correct implementations of synchronization constructs by low-level primitives, our starting point is to view the implementation 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 main correctness condition *with respect to an equational theory* [21, 22, 27], since it ensures that program transformations of the target calculus $\mathcal{C}'$ can be soundly applied to the translated $\mathcal{C}$ programs. Adequacy can be defined for any kind of program semantics that gives rise to a preorder $\leq_{\mathcal{C}}$ (or an equivalence relation $=_{\mathcal{C}}$, that may be declared as $\leq_{\mathcal{C}} \cap \geq_{\mathcal{C}}$ if necessary) on the programs of $\mathcal{C}$, e.g. a denotational, bisimulation-based, or operationally-defined observational semantics. Formally, a translation is adequate if all (equally typed) programs with equivalent translations are equivalent, i.e., $T(p_1) \leq_{\mathcal{C}'} T(p_2)$ implies $p_1 \leq_{\mathcal{C}} p_2$. If additionally the converse holds, then T is called fully abstract.

For correctness *with respect to the observations* in the domain and co-domain of a translation we use the notion of *observational correctness*, which holds if the observations are not changed by the translation. Observational correctness implies adequacy and also – under mild conditions – that the translation is a fully abstract translation into the image-calculus.

In this paper, we assume observational (i.e. contextual) semantics based on operationally-defined forms of may- and must-convergence [4, 13, 3]. Our form of must-convergence is similar to the should-testing of Rensink and Vogler [18]. A common feature is that fairness of execution is mirrored in the semantic theory. This combination of may- and must-convergence properly captures the nondeterminism arising in concurrent programming languages [11, 24].

We investigate the lambda calculus with futures [12, 11], a formalization of the operational semantics of Alice ML [23]. We start with an enriched core language of Alice ML, the calculus $\lambda^\tau(\mathsf{fch})$. This is a **t**yped call-by-value lambda with **f**utures, polymorphic data and type **c**onstructors, concurrent threads, reference cells, and **h**andled futures which support single assignment of values to futures by concurrent threads. Futures are the synchronization primitive of Alice ML.

In previous work [11] we analyzed a less expressive untyped core language (the calculus $\lambda(\mathsf{f'h})$) which lacks data constructors and case expressions. There, we proved a rich set of program transformations correct wrt. contextual equivalence using diagram-based techniques based on the operational semantics. Instead of applying this technically involved and complex mechanism again to $\lambda^\tau(\mathsf{fch})$, we use adequacy to lift the correctness results obtained for $\lambda(\mathsf{f'h})$ to $\lambda^\tau(\mathsf{fch})$, by finding suitable adequate translations. A small technical obstacle for this lifting is that $\lambda(\mathsf{f'h})$ uses a sharing variant of beta-reduction. Hence we also need to show that an adapted calculus $\lambda(\mathsf{fh})$ with (non-sharing) beta-reduction can be fully abstractly encoded in $\lambda(\mathsf{f'h})$.

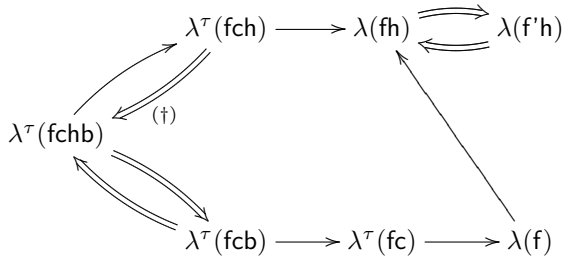
Equipped with these results we focus on the correctness of implementations of buffers in $\lambda^\tau(\mathsf{fch})$. To obtain a specification of buffers that is sufficiently rigorous for a correctness argument we extend $\lambda^\tau(\mathsf{fch})$ by concurrent **b**uffers (as e.g. used for implementing buffered channels in [16]), resulting in the calculus $\lambda^\tau(\mathsf{fchb})$. We then present an implementation of buffers into the buffer-free calculus, such that blocking buffers are translated into queuing and waiting, and waiting is translated into blocking. After formalizing this implementation as a translation we show adequacy of the translation. Moreover, the translation turns out to be observationally correct. As mentioned above, this means that the specification and implementation of buffers give rise to the same observations (in particular, they have the same convergence behavior).

We strengthen our result by showing that it is also possible to go in the opposite direction and correctly implement handled futures with buffers. In this case, the specification is again the calculus $\lambda^\tau(\mathsf{fchb})$, but now viewed as an extension of a handle-free calculus with buffers, called $\lambda^\tau(\mathsf{fcb})$. We provide a translation from $\lambda^\tau(\mathsf{fchb})$ to $\lambda^\tau(\mathsf{fcb})$ and show that it is fully abstract.

The implementations of buffers and handles lead to the question whether the constructs can already be encoded in a base language containing neither buffers nor handles. We show that this is indeed the case: buffers of $\lambda^\tau(\mathsf{fcb})$ can be encoded into the calculus $\lambda^\tau(\mathsf{fc})$, the calculus with futures without handles and buffers, using a fully abstract translation. This implementation of buffers relies on the presence of reference cells with atomic exchange operation in an essential way. In contrast to the previously mentioned encodings, it results in busy-wait situations, and is thus of a more low-level character than the other encodings.

The following diagram summarizes our results. Doubly lined arrows ($\Longrightarrow$) indicate fully abstract translations, while single lined arrows ($\longrightarrow$) indicate adequate translations. For completeness we also mention the calculus $\lambda(\mathsf{f})$ which is

like $\lambda(\mathsf{fh})$ without handled futures.



In fact, all these translations are observationally correct and thus also preserve the convergence behavior.

Of course there are additional encodings which are implied by the ones shown in the diagram. In particular, we mention that we obtain adequate encodings from $\lambda^\tau(\mathsf{fcb})$ into $\lambda^\tau(\mathsf{fch})$ and vice versa, by composing other translations. However, the proof of adequacy is indirect and facilitated by using the calculus $\lambda^\tau(\mathsf{fchb})$ – we expect a direct proof to be considerably more involved. We also note that the identity translation $\lambda^\tau(\mathsf{fch}) \xrightarrow{(\dagger)} \lambda^\tau(\mathsf{fchb})$ cannot be composed with the translation $\lambda^\tau(\mathsf{fchb}) \to \lambda^\tau(\mathsf{fch})$. The reasons for this are subtle, and explained in Remark 4.14. Nevertheless (†) can be composed with the fully abstract translation $\lambda^\tau(\mathsf{fchb}) \to \lambda^\tau(\mathsf{fcb})$, resulting in a fully abstract translation from $\lambda^\tau(\mathsf{fch}) \to \lambda^\tau(\mathsf{fcb})$.

For proving adequacy and full abstraction of the various encodings we rely on compositionality [26, 27], on commutation methods in order to prove invariants of implicit "queuing mechanisms" when implementing buffers via handles, as well as on equivalences for $\lambda^\tau(\mathsf{fch})$ that we lift from $\lambda(\mathsf{fh})$. In turn, we inherit equations for $\lambda^\tau(\mathsf{fchb})$ from $\lambda^\tau(\mathsf{fch})$ via the adequacy of this encoding.

Although our investigation concentrates on call-by-value lambda calculi with futures, we are confident that our methods and techniques can be applied to other (concurrent) core languages.

2 Lambda Calculus with Futures and Constructors

This section presents the calculus $\lambda^\tau(\mathsf{fch})$ underlying Alice ML. This is a typed lambda calculus with algebraic data types, concurrent and handled futures, and reference cells, which is obtained from the calculus with futures of [12] by adding data constructors with recursive polymorphic type constructors.

Data and type constructors. Our encodings require n-tuples $\langle v_1, \ldots, v_n \rangle$ of all possible types $\tau_1 \times \cdots \times \tau_n$. For the sake of generality and uniformity, we keep the concrete signature of data and type constructors as a parameter. Such a signature $\Sigma = (\mathcal{K}, \mathcal{D})$ consists of a finite ranked set of type constructors $\kappa \in \mathcal{K}$ and a finite ranked set of data constructors $k \in \mathcal{D}$. We denote the arities of data and type constructors by $ar(\cdot) \geq 0$. Polymorphic types $\widehat{\tau}$ over Σ have the following abstract syntax, where α belongs to a fixed infinite set of type

$$
\begin{array}{rcl}
\tau \in \mathit{Type} & ::= & \mathsf{unit} \mid \mathsf{ref}\ \tau \mid \tau \to \tau \mid \kappa(\tau_1, \ldots, \tau_{ar(\kappa)}) \\
c \in \mathit{Const} & ::= & \mathbf{unit} \mid \mathbf{ref}^\tau \mid \mathbf{thread}^\tau \mid \mathbf{lazy}^\tau \mid \mathbf{handle}^\tau \\
\pi \in \mathit{Pat} & ::= & k^\tau(x_1, \ldots, x_{ar(k)}) \\
e \in \mathit{Exp} & ::= & x \mid c \mid \lambda x.e \mid e_1\ e_2 \mid \mathbf{exch}(e_1, e_2) \mid k^\tau(e_1, \ldots, e_{ar(k)}) \\
 & & \mid \mathbf{case}_\kappa\ e\ \mathbf{of}\ \pi_1 \Rightarrow e_1 \mid \ \ldots\ \mid \pi_m \Rightarrow e_m \quad (m > 0) \\
v \in \mathit{Val} & ::= & x \mid c \mid \lambda x.e \mid k^\tau(v_1, \ldots, v_{ar(k)}) \\
p \in \mathit{Proc} & ::= & p_1 \mid p_2 \mid (\nu x)p \mid x\,\mathsf{c}\,v \mid x \Leftarrow e \mid x \overset{susp}{\Longleftarrow} e \mid y\,\mathsf{h}\,x \mid y\,\mathsf{h}\,\bullet
\end{array}
$$

Fig. 1. Types, expressions and processes of $\lambda^\tau(\mathsf{fch})$

variables:

$$\widehat{\tau} \in \mathit{PolyType} ::= \alpha \mid \mathsf{unit} \mid \mathsf{ref}\ \widehat{\tau} \mid \widehat{\tau} \to \widehat{\tau} \mid \kappa(\widehat{\tau}_1, \ldots, \widehat{\tau}_{ar(\kappa)})$$

Monomorphic types $\tau \in \mathit{Type}$ are polymorphic types without variables. We assume a unique polymorphic type $\mathrm{upt}(k) \in \mathit{PolyType}$ for each data constructor $k \in \mathcal{D}$, that has the form $\widehat{\tau}_1 \to \ldots \to \widehat{\tau}_{ar(k)} \to \kappa(\alpha_1, \ldots, \alpha_{ar(\kappa)})$ where $\kappa \in \mathcal{K}$ and only $\alpha_1, \ldots, \alpha_{ar(\kappa)}$ may occur as type variables in $\widehat{\tau}_j$ for all $j = 1, \ldots, ar(k)$. The set $\mathcal{D}(\kappa)$ consists of all data constructors $k \in \mathcal{D}$ for which κ occurs in the target type of $\mathrm{upt}(k)$. We assume that $\mathcal{D}(\kappa)$ is nonempty for all $\kappa \in \mathcal{K}$. In typing rules, we will write $\tau \preceq \widehat{\tau}$ if τ is a monomorphic instance of the polymorphic type $\widehat{\tau} \in \mathit{PolyType}$.

For instance, we can define lists of all types, when having a type constructor $\mathsf{List} \in \mathcal{K}$ with two data constructors $\mathsf{cons}, \mathsf{nil} \in \mathcal{D}(\mathsf{List})$ such that $\mathrm{upt}(\mathsf{cons}) = \alpha \to \mathsf{List}(\alpha) \to \mathsf{List}(\alpha)$ and $\mathrm{upt}(\mathsf{nil}) = \mathsf{List}(\alpha)$. For n-tuples (where $n \geq 0$), we assume type constructors $\cdot \times \cdots \times \cdot \in \mathcal{K}$ and data constructors $\langle \cdot, \ldots, \cdot \rangle \in \mathcal{D}$ of arity n, whose unique polymorphic type is $\alpha_1 \to \ldots \to \alpha_n \to (\alpha_1 \times \cdots \times \alpha_n)$.

Syntax of $\lambda^\tau(\mathsf{fch})$. We start from a signature $(\mathcal{K}, \mathcal{D})$, a set of variables *Var*, and a global assignment of variables to monomorphic types $\Gamma : \mathit{Var} \to \mathit{Type}$ such that for every $\tau \in \mathit{Type}$ there exists infinitely many $x \in \mathit{Var}$ with $\Gamma(x) = \tau$. Consistent renaming of variables must preserve the type. The syntax of $\lambda^\tau(\mathsf{fch})$ defined in Fig. 1 consists of two layers: a level of λ-expressions $e \in \mathit{Exp}$ 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. Expressions e subsume values v as usual in a call-by-value λ-calculus. Compared to the original lambda calculus with futures, we add constructor applications $k^\tau(e_1, \ldots, e_{ar(k)})$ creating data structures by constructors $k \in \mathcal{D}$ labeled with monomorphic types $\tau \in \mathit{Type}$ and typed case-expressions $\mathbf{case}_\kappa\ e\ \mathbf{of}\ \pi_1 \Rightarrow e_1 \mid \ \ldots\ \mid \pi_m \Rightarrow e_m$ whose pattern are non-overlapping and exhaustive. I.e., every constructor $k \in \mathcal{D}(\kappa)$ appears exactly once in some pattern π_i. A pattern has the form $k^\tau(x_1, \ldots, x_{ar(k)})$ such that no variable appears twice. All variables in a pattern π of a branch $\pi \Rightarrow e$ are bound with scope in e. 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.

New components in expressions are introduced by typed (higher-order) constants. The constant $\mathbf{ref}^\tau$ introduces a reference cell. The constants $\mathbf{thread}^\tau$ and

$$p_1 \mid p_2 \;\equiv\; p_2 \mid p_1 \qquad (p_1 \mid p_2) \mid p_3 \;\equiv\; p_1 \mid (p_2 \mid p_3)$$
$$(\nu x)(\nu y)p \equiv (\nu y)(\nu x)p \qquad (\nu x)(p_1) \mid p_2 \equiv (\nu x)(p_1 \mid p_2) \quad \text{if } x \notin fv(p_2)$$

Fig. 2. Structural congruence of processes

$\mathbf{lazy}^\tau$ serve for introducing eager threads and lazy threads, each of them together with a future. Finally, the constant $\mathbf{handle}^\tau$ is used to generate futures with an associated handler. For convenience we sometimes omit the type label of constants as well as constructors if it is obvious or not important. 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.

As in the pi-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 binder. It can be seen as hiding variables, whereas free variables are visible for outside observers. 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 pi-calculus. Cells $x \,\mathsf{c}\, v$ associate (a memory location) x to a value v. 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. 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. A process p *introduces* a variable x if $p \equiv p_1$ or $p \equiv p_1 \mid p_2$ for p_1 a component of the following form (for some e, v and y): $x \,\mathsf{c}\, v$, or $x \Leftarrow v$, or $x \overset{susp}{\Longleftarrow} e$, or $y \,\mathsf{h}\, x$, or $x \,\mathsf{h}\, y$, or $x \,\mathsf{h}\, \bullet$. A process is *well-formed* if no subprocess introduces any variable more than once. For instance, neither $x \Leftarrow v \mid x \,\mathsf{c}\, v'$ nor $(\nu x)(x \Leftarrow v \mid x \,\mathsf{c}\, v')$ is well-formed.

Typing. In order to have a consistent notion of typed program transformation, we rely on unique monomorphic typings. To this end, we already assumed a unique type $\Gamma(x)$ for all variables. For expressions, we assign types in judgements $e : \tau$. Process components have to be well-typed, written $p : wt$. The typing rules for expressions and processes can be found in Fig. 3 and 4. Note that the well-formedness conditions for processes described earlier are kept orthogonal to typing, in contrast to the type system of [12]. We write $e[e'/x]$ for the (capture-free) substitution of x by e' in e. It preserves the type of e if $e' : \Gamma(x)$ holds.

Syntactic Abbreviations. We assume that the set of type constructors contains a nullary constructor $\mathsf{bool} \in \mathcal{K}$, with nullary data constructors true and false. For convenience, we will freely use the usual syntactic 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 $\lambda x.\mathbf{case}\ x\ \mathbf{of}\ \pi \Rightarrow e; \pi' \Rightarrow z$ etc. (where z represents an error, and for instance can be defined by the component $z \Leftarrow z$). Instead of $\mathbf{case}\ e\ \mathbf{of}\ \mathsf{true} \Rightarrow e_1 \mid \mathsf{false} \Rightarrow e_2$ we write $\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$,

$$\frac{}{\mathbf{unit} : \mathsf{unit}} \qquad \frac{\tau \preceq \alpha{\to}\mathsf{ref}\ \alpha}{\mathbf{ref}^{\tau} : \tau} \qquad \frac{\tau \preceq (\alpha{\to}\alpha){\to}\alpha}{\mathbf{thread}^{\tau} : \tau} \qquad \frac{\tau \preceq (\alpha{\to}\alpha){\to}\alpha}{\mathbf{lazy}^{\tau} : \tau}$$

$$\frac{\tau \preceq (\alpha_1{\to}(\alpha_1{\to}\mathsf{unit}){\to}\alpha_2){\to}\alpha_2}{\mathbf{handle}^{\tau} : \tau}$$

$$\frac{\Gamma(x) = \tau}{x : \tau} \qquad \frac{x{:}\tau_1 \quad e : \tau_2}{(\lambda x.e) : \tau_1 \to \tau_2} \qquad \frac{e_1 : \tau_1 \to \tau_2 \quad e_2 : \tau_1}{(e_1\ e_2) : \tau_2} \qquad \frac{e_1 : \mathsf{ref}\ \tau \quad e_2 : \tau}{\mathbf{exch}(e_1, e_2) : \tau}$$

$$\frac{k \in \mathcal{D}(\kappa) \quad \tau = \tau_1 \to \ldots \to \tau_{ar(k)} \to \kappa(\tau'_1, \ldots, \tau'_{ar(\kappa)}) \preceq \mathrm{upt}(k) \quad \forall j \in 1 \ldots ar(k).\ e_j : \tau_j}{k^{\tau}(e_1, \ldots, e_{ar(k)}) : \kappa(\tau'_1, \ldots, \tau'_{ar(\kappa)})}$$

$$\frac{\begin{array}{c}\mathcal{D}(\kappa) = \{k_1, \ldots, k_n\} \quad e : \kappa(\tau'_1, \ldots, \tau'_{ar(\kappa)}) \quad \forall i = 1 \ldots n.\ \ e_i : \tau \\ \forall i = 1 \ldots n.\ \tau_i = \Gamma(x_{i,1}) \to \ldots \to \Gamma(x_{i,ar(k_i)}) \to \kappa(\tau'_1, \ldots, \tau'_{ar(\kappa)}) \preceq \mathrm{upt}(k_i)\end{array}}{(\mathbf{case}_{\kappa}\ e\ \mathbf{of}\ (k_i^{\tau_i}(x_{i,1}, \ldots, x_{i,ar(k_i)})\ \Rightarrow e_i)^{i=1\ldots n}) : \tau}$$

Fig. 3. Types of expressions

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

$$\frac{}{y\ \mathsf{h}\ \bullet : wt} \qquad \frac{x{:}\tau \qquad y{:}\tau \to \mathsf{unit}}{y\ \mathsf{h}\ x : wt}$$

Fig. 4. Well-typed processes.

and the special case $\mathsf{if}\ e\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ \mathsf{true}$ is written as $\mathsf{wait}\ e$. The symbol '_' stands for an arbitrary fresh variable. Finally, we write newhandled as shorthand for $\mathbf{handle}\ \lambda f \lambda h.\langle h, f\rangle$.

Contexts and Operational Semantics. The operational semantics defines an evaluation strategy via evaluation contexts in which reduction rules apply. We introduce *contexts* C and D. An *expression context* C is a process where exactly one expression-position is replaced with a typed hole marker $[\cdot]^{\tau}$. For technical reasons it is important that this position is not syntactically restricted to just values: e.g., in a cell $x\ \mathsf{c}\ (k(\mathsf{true}, \lambda y.e))$ the position of a hole can only be within e. A *process context* D is a process where exactly one process position is replaced with the hole marker $[\cdot]$.

We only consider well-typed contexts, where the typing of contexts is like the typing of expressions, with two additional typing rules for the context hole:

$$\frac{}{[\cdot]^{\tau} : \tau} \qquad \frac{}{[\cdot] : wt}$$

The result of placing the expression $e : \tau$ (process p, resp.) in context C with hole $[\cdot]^{\tau}$ (context D, resp.), possibly capturing free variables of e (p, resp.), is written $C[e]$ ($D[p]$, resp.). It is easy to verify that $D[p] : wt$ if $D : wt$ and $p : wt$,

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

Fig. 5. Evaluation contexts

and that for expression contexts $C[[\cdot]^\tau] : wt$ and expressions $e : \tau$, $C[e] : wt$ holds.

Fig. 5 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. 6. We write $\xrightarrow{\mathsf{ev}}$ for $\to$ when we want to distinguish reductions from *transformations* below. We use $\xrightarrow{*}$ for the reflexive-transitive closure of $\to$ and $\xrightarrow{+}$ for the transitive closure. We sometimes label reductions with their name, e.g. $\xrightarrow{\text{BETA}(\mathsf{ev})}$, and the notation $p \xrightarrow{a \vee b} q$ means that either $p \xrightarrow{a} q$ or $p \xrightarrow{b} q$ holds.

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 its previous contents. Since this is an atomic operation, no other thread can interfere. 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)) creates handle components. 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 $x \Leftarrow v$. In particular, notice that a used handle component $y\,\mathsf{h}\,\bullet$ indicates that y is a handle that has already been used to bind its associated future. It gets stuck if an expression $E[y\ v]$ has to be evaluated. Type safety holds for reduction, i.e., reduction preserves well-typedness (and well-formedness) of processes.

As an example, let r:(ref $\mathsf{List}(\tau)$) be a reference cell containing τ-lists. The effect of reducing process $r\,\mathsf{c}\,v \mid (\nu x)(x \Leftarrow \mathbf{thread}\ \lambda y.(\mathbf{exch}(r, \mathsf{cons}(v', y))))$ is the substitution of the cell content v by $\mathsf{cons}(v', v)$ in a *single atomic step*. More precisely, it is $(\nu x)(\nu y)(r\,\mathsf{c}\,\mathsf{cons}(v', y) \mid x \Leftarrow y \mid y \Leftarrow v)$. This process is observationally equivalent to $r\,\mathsf{c}\,\mathsf{cons}(v', v)$, as we will show later on.

As a second example, suppose ack is a (nullary) type constructor with a single data constructor Ack, and assume p is a thread suspending on a handled future x of type ack: $p \equiv F[\mathbf{case}\ x\ \mathbf{of}\ \mathsf{Ack} \Rightarrow e]$. The thread may resume the computation

Reduction rules.

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

Well-formed processes. The rules can only be applied to well-formed processes.

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 above). 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. 6. Small-step reduction relation of $\lambda^{\tau}(\mathsf{fch})$, denoted by $\to$ or $\overset{\mathsf{ev}}{\to}$

of e once a second thread uses the associated handler and provides a value for x: the process $p \mid F'[h\,\mathsf{Ack}] \mid h\,\mathsf{h}\,x$ can reduce to $F[e] \mid F'[\mathbf{unit}] \mid x{\Leftarrow}\mathsf{Ack} \mid h\,\mathsf{h}\,\bullet$, after (HANDLE.BIND(ev)), (FUT.DEREF(ev)) and (CASE.BETA(ev)) reductions. In this way, handled futures are the basic synchronization construct in $\lambda^{\tau}(\mathsf{fch})$.

Observations and Contextual Equivalence. A process p is *successful* if it is well-formed and in every component $x{\Leftarrow}e$ of p, the identifier x is bound possibly via a chain $x{\Leftarrow}x_1 \mid x_1{\Leftarrow}x_2 \mid \ldots \mid x_{n-1}{\Leftarrow}x_n \mid x_n{\Leftarrow}v$ to a non-variable value, a cell or a lazy future, a handle, or a handled future. Hence, in a non-failing computation, every non-lazy 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) and $x{\Leftarrow}(\lambda u.\lambda v.v)\,(y\,\mathbf{unit}) \mid y{\Leftarrow}(\lambda u.\lambda v.v)\,(x\,\mathbf{unit})$ (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 $p{\uparrow} \Leftrightarrow \neg p{\Downarrow}$. Now, for $\xi \in \{{\downarrow}, {\Downarrow}\}$, we define

contextual approximation relations between processes p_1 and p_2 by:

$$p_1 \leq_\xi p_2 \quad \Leftrightarrow \quad \forall D:\ D[p_1]\xi \Rightarrow D[p_2]\xi$$

We write $p_1 \leq p_2$ if both $p_1 \leq_\downarrow p_2$ and $p_1 \leq_\Downarrow p_2$ hold, and $p_1 \sim p_2$ if both $p_1 \leq p_2$ and $p_2 \leq p_1$ hold. The same definitions for expressions e_1, e_2 of equal type τ and expression contexts $C[[\cdot]^\tau]$ results in relations $\leq_{\downarrow,\tau}$, $\leq_{\Downarrow,\tau}$, and $\sim_\tau$.

Translations. We recall the framework of [27], where an abstract calculus $\mathcal{C}$ consists of sets of (well-typed) processes p, contexts D, and convergence predicates ξ. The calculus $\lambda^\tau(\mathsf{fch})$ and the other (possibly untyped) calculi introduced in the subsequent sections fit into this general framework. A *translation T between two such calculi* maps well-typed processes to well-typed processes, and contexts to contexts. A translation T between calculi $\mathcal{C}$ and $\mathcal{C}'$ is *convergence equivalent* if $T(p)\xi \Leftrightarrow p\xi$ for all p and all convergence predicates ξ. The translation T is *compositional* iff for all contexts D and processes p we have $T(D)[T(p)] = T(D[p])$. A translation T is *observationally correct*, if for all all programs p and all contexts D, and all convergence predicates ξ: $T(D[p])\xi \iff T(D)[T(p)])\xi$. A translation is *adequate* if T reflects operational approximation, i.e. if $T(p_1) \leq^{\mathcal{C}'} T(p_2) \Rightarrow p_1 \leq^{\mathcal{C}} p_2$ for all p_1, p_2. Finally, if T additionally preserves inequations, i.e. if $T(p_1) \leq^{\mathcal{C}'} T(p_2) \Leftrightarrow p_1 \leq^{\mathcal{C}} p_2$ for all p_1, p_2 holds, then it is *fully abstract*.

As described in the introduction, adequacy and full abstraction relate to the equational theories of the source and target language of a translation, and adequate translations provide useful tools for transferring equations. The soundness of an encoding, in the sense that each program is indistinguishable from its translation, is given by observational correctness. Of course, these notions are related:

Proposition 2.1 (Adequacy, [27]). *If a translation T is compositional and convergence equivalent, then T is adequate and observationally correct. Moreover, observational correctness of a translation implies adequacy of the translation.*

For the considered calculi in this paper we also require compositionality on expressions, i.e. $T(C[e]) = T(C)[T(e)]$ and type correctness of T for expressions. These requirements simplify the corresponding proofs for processes. Moreover, they enable us to derive equivalences from the adequacy of the translations not only between processes but also between expressions, i.e. that $T(e) \sim_{T(\tau)} T(e')$ implies $e \sim_\tau e'$.

It is easy to verify that translations compose:

Proposition 2.2 (Composition). *Let $\mathcal{C}, \mathcal{C}', \mathcal{C}''$ be calculi, and $T : \mathcal{C} \to \mathcal{C}'$, $T' : \mathcal{C}' \to \mathcal{C}''$ be translations. Then $T' \circ T : \mathcal{C} \to \mathcal{C}''$ is also a translation, and if T, T' are compositional (observationally correct, adequate, fully-abstract, respectively), then also the composition $T' \circ T$ is compositional (observationally correct, adequate, fully-abstract, respectively).*

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

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

Closure. Transformations are closed under $\equiv$ and D-contexts.

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

We also recall a criterion for fully abstract translations, which can be used if only new primitives are added to a calculus $\mathcal{C}'$. The statement of this criterion in [27] contains a flaw; the following corrected version is proved in the technical report [25].

Proposition 2.3 (Full abstraction for extensions).
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 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 ι is fully abstract.
If T is injective on types, then T is also fully abstract.

If a translation T is observationally correct and injective on types, then $\leq$ is retained under T, relative to its image.

Remark 2.4 (on full abstraction on images). A variation of this full abstraction result is possible [25]. Let $\mathcal{C}, \mathcal{C}'$ be calculi and $T : \mathcal{C} \to \mathcal{C}'$ be an observationally correct translation. Let $\mathcal{C}'' := T(\mathcal{C})$ be the subcalculus of $\mathcal{C}'$ consisting of the images under T, and let $\leq_T$ be the preorder defined on $\mathcal{C}''$. Moreover, assume that for all τ, T is surjective on the programs of type τ and for every τ', T is a surjective mapping $T : \mathcal{O}_{\tau_1,\tau_2} \to \mathcal{O}_{T(\tau_1),T(\tau_2)}$, where $\mathcal{O}_{\tau_1,\tau_2}$ are the contexts of type τ_2 with hole of type τ_1. Then for all types τ and programs p_1, p_2: $p_1 \leq_\tau p_2 \iff T(p_1) \leq_{T,T(\tau)} T(p_2)$. That is, the translation is fully abstract as translation $T : \mathcal{C} \to \mathcal{C}''$.

3 Correctness of Transformations in $\lambda^{\tau}(\mathsf{fch})$

In the correctness proofs we will make use of program transformations, which are called *correct* if whenever p_1 is transformed into p_2, then $p_1 \sim p_2$. In Fig. 7 some program transformations are defined. It is important that program transformations preserve the types of replaced subexpressions. E.g. the rule (β-CBV(a)) may also be applied from right to left, and in this case, we must choose a variable x

with $\Gamma(x) = \tau$ where τ is the (uniquely determined) type of the value v. The use of the framework sketched at the end of Section 2 makes it possible to lift process equivalences from the untyped lambda calculus with futures [11] to correct program transformations in $\lambda^\tau(\mathsf{fch})$. We establish these correctness results by using an adequate translation from $\lambda^\tau(\mathsf{fch})$ into $\lambda(\mathsf{f'h})$, which technically requires another intermediate calculus $\lambda(\mathsf{fh})$ which can be translated fully-abstractly into $\lambda(\mathsf{f'h})$.

In summary, in this section we prove the following part of the diagram from page 4, where *enc* is the translation removing case and constructors and *id* is the respective identity translation.

$$\lambda^\tau(\mathsf{fch}) \xrightarrow{enc} \lambda(\mathsf{fh}) \underset{id}{\overset{id}{\rightleftarrows}} \lambda(\mathsf{f'h})$$

In the subsequent section we show that the translations between $\lambda(\mathsf{fh})$ and $\lambda(\mathsf{f'h})$ are fully-abstract. Thus we can lift the known equivalences for $\lambda(\mathsf{f'h})$ into $\lambda(\mathsf{fh})$. In Section 3.2 we provide an encoding from $\lambda^\tau(\mathsf{fch})$ into $\lambda(\mathsf{fh})$ and show that this translation is adequate, which allows us at the end to lift program equivalences into $\lambda^\tau(\mathsf{fch})$.

3.1 Modifying beta(ev)-Reduction

The calculus $\lambda(\mathsf{f'h})$ described in [11] is the subcalculus of $\lambda^\tau(\mathsf{fch})$ without constructors, case-expressions, and typing. Moreover, instead of the β-CBV(ev)-reduction rule from Fig. 6, a sharing variant is 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 $E[(\lambda y.e)\, v] \to (\nu y)(E[e] \mid y \Leftarrow v)$.

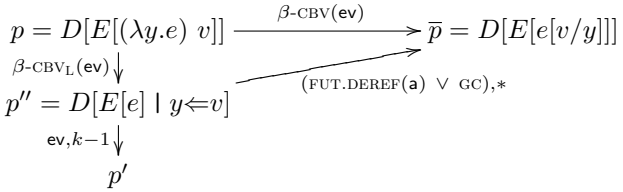
We call the modification of $\lambda(\mathsf{f'h})$ with the usual (substituting) call-by-value β-reduction $\lambda(\mathsf{fh})$ (this is the calculus presented in [12]). We first prove that the identity translation from $\lambda(\mathsf{fh})$ into $\lambda(\mathsf{f'h})$ is fully-abstract. This essentially means that, as far as contextual equivalence is concerned, it does not matter which β-rule we choose.

Obviously, the identity translation from $\lambda(\mathsf{f'h})$ to $\lambda(\mathsf{fh})$ as well as the identity translation from $\lambda(\mathsf{fh})$ to $\lambda(\mathsf{f'h})$ are bijective and compositional. We show that both translations are convergence equivalent, which by Proposition 2.3 implies that both translations are fully-abstract.

Lemma 3.1. *For all $\lambda(\mathsf{f'h})$-processes p, $p{\downarrow} \Leftrightarrow p{\downarrow_{\lambda(\mathsf{fh})}}$ and $p{\Downarrow} \Leftrightarrow p{\Downarrow_{\lambda(\mathsf{fh})}}$, i.e. the identity translations from $\lambda(\mathsf{f'h})$ and $\lambda(\mathsf{fh})$ and back are convergence equivalent.*

Proof. 1. Theorem 4.23 of [11] proves that β-CBV(a) is a correct program transformation for $\lambda(\mathsf{f'h})$. From $p{\downarrow_{\lambda(\mathsf{fh})}}$, induction on the length of a reduction from p to a successful process shows that $p{\downarrow}$. This is because the reduction corresponding to $p{\downarrow_{\lambda(\mathsf{fh})}}$ consists of $\lambda(\mathsf{f'h})$-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 $\lambda(\mathsf{f'h})$. We use induction on k. If $k = 0$, then p is also a successful process of $\lambda(\mathsf{fh})$. 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 $\lambda(\mathsf{fh})$, then the claim follows by using the induction hypothesis. If the reduction is a β-$\mathrm{CBV_L}(\mathsf{ev})$-reduction, then we have the following situation:



It is easy to verify that the sequence of (FUT.DEREF(a) ∨ 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 [11] for the transformation (FUT.DEREF(a)); for (GC) it follows from Lemma 4.7 in combination with Theorem 4.8 in [11].
3. Assume that $p\uparrow_{\lambda(\mathsf{fh})}$, and note that (1),(2) imply $p\Uparrow \Leftrightarrow p\Uparrow_{\lambda(\mathsf{fh})}$. 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 3.2. *The identity translations from* $\lambda(\mathsf{f'h})$ *into* $\lambda(\mathsf{fh})$ *and back are fully-abstract.*

This allows us to transfer correct transformations for $\lambda(\mathsf{f'h})$ shown in [11] to $\lambda(\mathsf{fh})$.

Corollary 3.3. *All reductions of* $\lambda(\mathsf{fh})$ *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* $\lambda(\mathsf{fh})$*.*

Proof. Correctness of the transformations was established for $\lambda(\mathsf{f'h})$ in [11]. Because of full abstraction, they are also correct in $\lambda(\mathsf{fh})$.

3.2 Removing Constructors and Types

In order to lift contextual equivalences from $\lambda(\mathsf{fh})$ to $\lambda^\tau(\mathsf{fch})$, we construct a translation $enc : \lambda^\tau(\mathsf{fch}) \to \lambda(\mathsf{fh})$ that is observationally correct and adequate. Note that for adequacy of this encoding it is necessary that $\lambda^\tau(\mathsf{fch})$ is typed, since otherwise untyped programs that get stuck due to a dynamic type error may become must-convergent after the translation. Furthermore, it is not possible to

restrict $\lambda(\mathsf{fh})$ to monomorphic typing, since the encoding of case and constructors cannot be monomorphically typed. These problems are already extensively discussed in [27] where illustrating examples can be found.

The main part is to encode the case and constructors, thus removing them, and to show that the translation has all required properties.

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

We provide an encoding $enc_1 : \lambda^\tau(\mathsf{fch}) \to \lambda^\tau(\mathsf{fch}_{av})$, where additionally we introduce some labels to mark the encoded constructors. These labels are used in later proofs. The encoding enc_1 uses for the syntactic correct translation of cell-values the encoding $enc\#_1$. Both translations are defined as follows:

$$\begin{aligned}
enc_1(x\,\mathsf{c}\,v) &\triangleq x\,\mathsf{c}\,enc\#_1(v)\\
enc_1(k(e_1,\ldots,e_n)) &\triangleq (\lambda^{l_1}x_1.\ \ldots\lambda^{l_n}x_n.k(x_1,\ldots,x_n))\ enc_1(e_1)^{l_1}\ldots enc_1(e_n)^{l_n},\\
&\qquad \text{where } x_i \text{ are fresh, and } l_i \text{ are new labels}\\
enc_1(t) &\triangleq \text{homomorphically wrt. the term structure of } t
\end{aligned}$$

$$\begin{aligned}
enc\#_1(\lambda x.e) &\triangleq \lambda x.enc_1(e)\\
enc\#_1(k(v_1,\ldots,v_n) &\triangleq k(enc\#_1(v_1),\ldots,enc\#_1(v_n))
\end{aligned}$$

The translation enc_1 (but not the translation $enc\#_1$) is 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 $\lambda^\tau(\mathsf{fch}_{av})$ to $\lambda(\mathsf{fh})$-processes, by removing constructors, case expressions, and types: Let $K = \mathcal{D}(\kappa)$ be the set of constructors for a specific type constructor κ and let $\tau_1,\ldots,\tau_m$ be types. By the assumptions on the signature, K is non-empty. We choose an arbitrary (but from now on fixed) order of the constructors in K, $k_1,\ldots,k_n$ where $n \geq 1$. A constructor application of $\lambda^\tau(\mathsf{fch}_{av})$ 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:

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

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, i.e. $\mathbf{thread}^\tau \triangleq \mathbf{thread}$, $\mathbf{ref}^\tau \triangleq \mathbf{ref}$, $\mathbf{lazy}^\tau \triangleq \mathbf{lazy}$, and $\mathbf{handle}^\tau \triangleq \mathbf{handle}$.

Definition 3.4. *The translation* $enc : \lambda^\tau(\mathsf{fch}) \to \lambda(\mathsf{fh})$ *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 $\mathbf{cons} : \tau \to \mathit{list}(\tau) \to \mathit{list}(\tau)$ and $\mathbf{nil} : \mathit{list}(\tau)$. Then:

$$\begin{aligned}
&\mathit{enc}(\mathbf{cons}(e_h, e_t)) && = \mathit{enc}_2(\mathit{enc}_1(\mathbf{cons}(e_h, e_t)))\\
&&& = \mathit{enc}_2(\lambda x_1, x_2\ (\mathbf{cons}(x_1, x_2))\ \mathit{enc}_1(e_h)\ \mathit{enc}_1(e_t))\\
&&& = (\lambda x_1, x_2, a_{nil}, a_{cons}.a_{cons}\ x_1\ x_2\ \mathbf{unit})\ \mathit{enc}(e_h)\ \mathit{enc}(e_t)\\
&\mathit{enc}(\mathbf{nil}) && = \mathit{enc}_2(\mathbf{nil}) = (\lambda a_{nil}, a_{cons}.a_{nil}\ \mathbf{unit})\\
&\mathit{enc}\begin{pmatrix}\mathbf{case}_{list}\ e\ \mathbf{of}\\ \quad \mathbf{nil} \Rightarrow e_1\\ \mid \mathbf{cons}(x, y) \Rightarrow e_2\end{pmatrix} && = \mathit{enc}(e)\ (\lambda_.\mathit{enc}(e_1))\ (\lambda x, y.\lambda_.\mathit{enc}(e_2))
\end{aligned}$$

3.3 Adequacy of enc_2

Lemma 3.5. *For every value v of $\lambda^\tau(\mathsf{fch}_{av})$, $enc_2(v)$ is a $\lambda(\mathsf{fh})$-value.*

Proof. Clear, since all values of the form $k(\ldots)$ are translated into abstractions.

Lemma 3.6. *For all $p_1, p_2 \in \lambda^\tau(\mathsf{fch}_{av})$ 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 $\lambda(\mathsf{fh})$: since only $\lambda^\tau(\mathsf{fch}_{av})$-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:

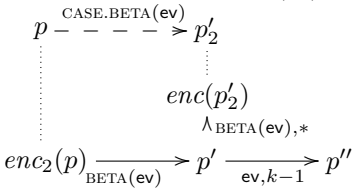
$$\begin{aligned}
&\quad \mathit{enc}_2(p_1)\\
&= \mathit{enc}_2(D[E[\mathbf{case}\ k_j(v_1, \ldots, v_{ar(k_j)})\ \mathbf{of}\ (k_i(x_1, \ldots, x_{ar(k_i)}) \Rightarrow e_i)^{i=1\ldots n}]])\\
&= \mathit{enc}_2(D)[\mathit{enc}_2(E)[\mathit{enc}_2(\mathbf{case}\ k_j(v_1, \ldots, v_{ar(k_j)})\\
&\qquad\qquad \mathbf{of}\ (k_i(x_1, \ldots, x_{ar(k_i)}) \Rightarrow e_i)^{i=1\ldots n}])]\\
&= \mathit{enc}_2(D)[\mathit{enc}_2(E)\left[\begin{array}{l}((\lambda p_1, \ldots, p_n.p_j\ \mathit{enc}_2(v_1)\ \ldots\ \mathit{enc}_2(v_{ar(k_j)})\ \mathbf{unit})\\ \quad (\lambda x_{1,1}, \ldots, x_{1,ar(k_1)}.\lambda_.\mathit{enc}_2(e_1))\\ \quad \ldots\\ \quad (\lambda x_{n,1}, \ldots, x_{n,ar(k_n)}.\lambda_.\mathit{enc}_2(e_n))\end{array}\right])]\\
&\xrightarrow{\beta\text{-CBV}(\mathsf{ev}),*} \mathit{enc}_2(D)[\mathit{enc}_2(E)[((\lambda x_{j,1}, \ldots, x_{j,ar(k_j)}.\\
&\qquad\qquad \lambda_.\mathit{enc}_2(e_j))\ \mathit{enc}_2(v_1) \ldots \mathit{enc}_2(v_{ar(k_j)})\ \mathbf{unit})]]\\
&\xrightarrow{\beta\text{-CBV}(\mathsf{ev}),*} \mathit{enc}_2(D)[\mathit{enc}_2(E)[\mathit{enc}_2(e_j)[\mathit{enc}_2(v_1)/x_{j,1}, \ldots, \mathit{enc}_2(v_{ar(k_j)})/x_{j,ar(k_j)}]]]\\
&= \mathit{enc}_2(p_2)
\end{aligned}$$

Lemma 3.5 ensures that all $enc_2(v_i)$ are values, validating the β-CBV(ev) steps.

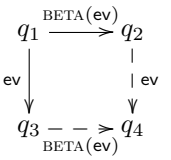
Proposition 3.7. *The translation enc_2 is observationally correct and adequate.*

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 3.5 shows that $enc_2(p)$ is successful when p is. The induction step follows by the simulation in Lemma 3.6.
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, 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 free. In 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 $\lambda(\mathsf{fh})$: $enc_2(p) \xrightarrow{\text{BETA}(\mathsf{ev}),*} p_2$, $p \xrightarrow{\text{CASE.BETA}(\mathsf{ev})} p_2'$, 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 $\lambda(\mathsf{fh})$: If $q \xrightarrow{\mathsf{ev},k} q'$ where q' is successful and $q \xrightarrow{\text{BETA}(\mathsf{ev})} q''$ 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 $\leq k$. The induction hypothesis implies $p_2'\!\downarrow$ 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 3.8. *The following transformations are correct for $\lambda^{\tau}(\mathsf{fch}_{av})$:*

- *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 $\lambda^{\tau}(\mathsf{fch}_{av})$).*

Proof. Let $\approx$ be one of the transformations. Due to adequacy of enc_2 it suffices to show $enc_2(p) \sim enc_2(p')$ for every $p \approx p'$ where p, p' are well-typed and well-formed. This follows from Corollary 3.3 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.

3.4 Adequacy of enc_1

Let $\overline{enc_1}$ be the backtranslation from labeled $\lambda^\tau(\mathsf{fch}_{av})$-processes into $\lambda^\tau(\mathsf{fch})$-processes, which is homomorphic except for the case of labeled 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 3.9. *Let $p \in \lambda^\tau(\mathsf{fch}_{av})$ be a labeled process. We say that the labeling of p is* valid *iff the following conditions hold:*

- *if there is a labeled 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 labeled λ labeled with l_i, then the abstraction is of the following form $\lambda^{l_j} \ldots . \lambda^{l_i} x_i.\ \ldots\ \lambda^{l_n}.k(e_1, \ldots, e_{j-1}, x_j, \ldots, x_n)$ and there are $n - j + 1$ arguments of a nested application and the arguments are labeled with $j, j + 1, \ldots, n$.*

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

- *If a labeled 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 $p \in \lambda^\tau(\mathsf{fch})$ its encoding $enc_1(p)$ is validly labeled.

Lemma 3.10. *For all $p \in \lambda^\tau(\mathsf{fch})$: $\overline{enc_1}(enc_1(p)) = p$ and for all validly labeled $p \in \lambda^\tau(\mathsf{fch}_{av})$: $enc_1(\overline{enc_1}(p)) \xrightarrow{\beta\text{-CBV}(\mathsf{a}),*} p$.*

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 unlabeled 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 $\lambda^\tau(\mathsf{fch}_{av})$-value, β-CBV(a) can be used to reduce this further to $k(v_1, \ldots, v_n)$.

Lemma 3.11. *For all processes of $\lambda^\tau(\mathsf{fch}_{av})$: $p \xrightarrow{\mathsf{ev}} p' \Rightarrow p' \in \lambda^\tau(\mathsf{fch}_{av})$. Moreover, if p is validly labeled, then p' is validly labeled.*

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

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

Lemma 3.12. *Let p be a validly labeled process, and $p \xrightarrow{\mathsf{ev}} p'$.*

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

Proof. The first part follows, since the a-labeled 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-labeled, then the position of the reduction for p is moved to another position in $\overline{enc_1}(p)$. Nevertheless, since the labeling 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 3.13. *Let $p_0 \in \lambda^{\tau}(\mathsf{fch}_{av})$ be a validly labeled 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-labeled, then $\overline{enc_1}(p_1) = \overline{enc_1}(p_0)$ and the claim holds. Otherwise, the reduction is b-labeled or unlabeled. Then Lemma 3.12 shows that $\overline{enc_1}(p_0) \xrightarrow{\mathsf{ev}} \overline{enc_1}(p_1)$. Hence, the claim holds.

Corollary 3.14. *For all p, $enc_1(p)\!\downarrow \Rightarrow p\!\downarrow$.*

Proof. Assume $enc_1(p)\!\downarrow$. Lemma 3.13 shows $\overline{enc_1}(enc_1(p))\!\downarrow$, and thus $p\!\downarrow$.

Lemma 3.15. *Let v be a $\lambda^{\tau}(\mathsf{fch})$-value. Then there exists a $\lambda^{\tau}(\mathsf{fch}_{av})$-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-labeled reductions are necessary to obtain values from the translation of $enc_1(k(v_1 \ldots, v_{ar(k)}))$.

Lemma 3.16. *For the translation enc_1 the following holds:*

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

- *For every evaluation context* $\widetilde{E}$ *of* $\lambda^\tau(\mathsf{fch})$ *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* $\lambda^\tau(\mathsf{fch})$*, there exists a future evaluation context* $\widetilde{F}'$ *such that for all expressions* e*, process contexts* D*, 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 3.15 to obtain values from translated values.

Lemma 3.17. *For all* $p_1, p_2 \in \lambda^\tau(\mathsf{fch})$ *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 3.16, and inspection of all standard reduction rules.

Lemma 3.18. *For all processes* p *of* $\lambda^\tau(\mathsf{fch})$*,* $p{\downarrow} \Rightarrow enc_1(p){\downarrow}$.

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 3.15. These transformations are correct by Corollary 3.8, 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 3.17 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 3.8) we have $p_2{\downarrow}$ and thus $p_1{\downarrow}$. Applying correctness of β-CBV(a) again yields $enc_1(p){\downarrow}$.

Lemma 3.19. *For all validly labeled processes* p *of* $\lambda^\tau(\mathsf{fch}_{av})$*,* $p{\uparrow} \Rightarrow \overline{enc_1}(p){\uparrow}$.

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 3.18 $enc_1(\overline{enc_1}(p)){\downarrow}$. From Lemma 3.10 we have $enc_1(\overline{enc_1}(p)) \xrightarrow{\beta\text{-CBV}(\mathsf{a}),*} p$. Correctness of β-CBV(a) (Corollary 3.8) shows the required $p{\downarrow}$. The induction step is analogous to Lemma 3.13.

Corollary 3.20. *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 3.19 since $\overline{enc_1}(enc_1(p)) = p$ by Lemma 3.10. □

Proposition 3.21. *The translation* enc_1 *is observationally correct and adequate.*

Proof. Whenever $enc\#_1$ is used for the translation enc_1, at the corressonding position a context hole is disallowed, since these are value restricted positions. Hence, enc_1 is compositional. It remains to show convergence equivalence. We

have to show that may- and must-convergence are preserved and reflected by enc_1. From Corollary 3.14, Lemma 3.18, and Corollary 3.20 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 3.14, and the induction step is analogous to Lemma 3.18.

Since composition preserves observational correctness and adequacy, Propositions 3.7 and 3.21 imply:

Theorem 3.22. *The translation enc is observationally correct and adequate.*

Remark 3.23. Note that *enc* is not fully abstract. We give an example without proof: The expressions $\lambda x^{\mathsf{bool}}.x$ and $(\lambda x^{\mathsf{bool}}.\mathsf{if}\ x\ \mathsf{then}\ x\ \mathsf{else}\ x)$ are equivalent, but the translated expressions are not equivalent since they behave differently when applied to **unit**. The second expression is translated into $\lambda x.x\ x\ x$. Applying both expressions to **unit** will result in **unit** and in (**unit unit unit**). The first is a value, and the second is not a value.
This in particular shows that Proposition 2.3 is not applicable: there is no identity embedding from $\lambda(\mathsf{fh})$ into $\lambda^{\tau}(\mathsf{fch})$ since the former is untyped.

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

Theorem 3.24. *All of the following are correct transformations for* $\lambda^{\tau}(\mathsf{fch})$*:*

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

Proof. We must show that $p \sim p'$ for every $p \approx p'$ where p, p' are well-typed and well-formed and where $\approx$ is any of the listed transformations. By adequacy and the fact that enc_1 acts as identity on types, it suffices to prove that $enc_1(p), enc_1(p')$ are well-typed and well-formed. But this is guaranteed by the correctness results for $\lambda^{\tau}(\mathsf{fch}_{av})$ (Corollary 3.8) and using Lemma 3.16 to obtain values, Lemma 3.15 to obtain process, evaluation and future evaluation contexts in the image of enc_1.

4 Concurrent Buffers are Encodable in $\lambda^{\tau}(\mathsf{fch})$

By extending the syntax and operational semantics of $\lambda^{\tau}(\mathsf{fch})$, we provide a specification of one-place buffers that describes their desired behavior.

The calculus $\lambda^{\tau}(\mathsf{fch})$ is extended by new primitives for concurrent buffers. This defines the calculus $\lambda^{\tau}(\mathsf{fchb})$, with the syntactic extensions shown in Fig. 8. $\lambda^{\tau}(\mathsf{fchb})$ 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 new constants $\mathbf{buffer}^{\tau}$ to create a new buffer and $\mathbf{get}^{\tau}$ 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. Contexts C are as before, but extended to

Syntactic extensions:

$\tau \in \mathit{Type} ::= \mathsf{buf}\ \tau \mid \ldots$ $\quad$ $c \in \mathit{Const} ::= \mathbf{buffer}^\tau \mid \mathbf{get}^\tau \mid \ldots$

$e \in \mathit{Exp} ::= \mathbf{put}(e_1, e_2) \mid \ldots$ $\quad$ $p \in \mathit{Proc} ::= x\,\mathsf{b}\,- \mid x\,\mathsf{b}\,v \mid \ldots$

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

Extensions of the type system:

$$\frac{\tau \preceq \mathsf{unit} \to \mathsf{buf}\ \alpha}{\mathbf{buffer}^\tau : \tau} \qquad \frac{\tau \preceq \mathsf{buf}\ \alpha \to \alpha}{\mathbf{get}^\tau : \tau} \qquad \frac{e_1 : \mathsf{buf}\ \tau \quad e_2 : \tau}{\mathbf{put}(e_1, e_2) : \mathsf{unit}} \qquad \frac{}{x\,\mathsf{b}\,- : wt} \qquad \frac{x{:}\mathsf{buf}\,\tau \quad v : \tau}{x\,\mathsf{b}\,v{:}wt}$$

Extensions of the reduction rules:

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

Fig. 8. Extensions of $\lambda^\tau(\mathsf{fch})$ for $\lambda^\tau(\mathsf{fchb})$

the new syntax, such that exactly one expression-position, which is not restricted to values, is replaced with a typed hole marker $[\cdot]^\tau$.

Fig. 8 also summarizes the operational interpretation of the new constructs, and 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 (instances of) type schemes (see Fig. 8); type safety then extends to the calculus $\lambda^\tau(\mathsf{fchb})$. Contextual preorder is defined as expected: the notion of a *successful process* from $\lambda^\tau(\mathsf{fch})$ is extended so that $\lambda^\tau(\mathsf{fchb})$ also allows $x\,\mathsf{b}\,-$ and $x\,\mathsf{b}\,v$ as components of successful processes. A $\lambda^\tau(\mathsf{fchb})$ process is *well-formed* if (in addition to the other process variables) no buffer variables are introduced twice.

4.1 Implementing Buffers Using Handled Futures

In the rest of this section we will show that there is an observationally correct (and thus also adequate) translation $T_B : \lambda^\tau(\mathsf{fchb}) \to \lambda^\tau(\mathsf{fch})$ which encodes buffers by handled futures. Note that the proof of adequacy of T_B requires equivalences in $\lambda^\tau(\mathsf{fch})$, which are derived from [11] (see Theorem 3.24).

Any concrete realization of buffers will rely on (more or less intricate) non-interference properties and the preservation of various invariants. We consider a particular implementation of buffers in $\lambda^\tau(\mathsf{fch})$, in terms of reference cells, futures and handles. This induces a translation from $\lambda^\tau(\mathsf{fchb})$ into $\lambda^\tau(\mathsf{fch})$.

The implementation in $\lambda^\tau(\mathsf{fch})$ of operations corresponding to **buffer**, **put**, and **get** is shown in Fig. 9. The buffer data structure is 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

$$
\begin{array}{ll}
 & \mathit{buffer} \triangleq \lambda\_.\, \mathsf{let}\ \langle h, f\rangle = \mathsf{newhandled}, \langle h', f'\rangle = \mathsf{newhandled}, \\
 & \qquad \mathit{putg} = \mathbf{ref}(\mathsf{true}), \mathit{getg} = \mathbf{ref}(f), \\
 & \qquad \mathit{stored} = \mathbf{ref}(f'), \mathit{handler} = \mathbf{ref}(h) \\
(1) & \quad \mathsf{in}\ \mathbf{thread}\ \lambda\_.\langle \mathit{putg}, \mathit{getg}, \mathit{stored}, \mathit{handler}\rangle\ \mathsf{end}
\end{array}
$$

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

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

Fig. 9. Implementing the buffer operations *buffer*, *put* and *get*. The numbers (1), (2), (3) indicate subexpressions for later reference.

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 ref τ, 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 $\langle \mathit{putg}, \mathit{getg}, \mathit{stored}, \mathit{handler}\rangle$:

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

The procedure *buffer* yields a tuple representing an empty buffer, satisfying the invariants. The procedure *put*, when applied to a buffer $\langle \mathit{putg}, \mathit{getg}, \mathit{stored}, \mathit{handler}\rangle$ 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.

4.2 Implementation as Translation

The implementation gives rise to a *translation* T_B from $\lambda^\tau(\mathsf{fchb})$ into $\lambda^\tau(\mathsf{fch})$: **put**, **get**, and **buffer** are replaced by the resp. program code, *put*, *get*, and *buffer* from Fig. 9, where for **put**, the two arguments are translated into a pair. On process level, we replace:

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

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

These mappings are compatible with typing:

Lemma 4.1 (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 $e : \tau$ then $T_B(e) : T_B(\tau)$.*
2. *If p is well-typed, then $T_B(p)$ is well-typed.*
3. *If p is well-formed, then $T_B(p)$ is well-formed.*
4. *For $C[[\cdot]^\tau] : wt$ and $D : wt$ we have $T_B(C[[\cdot]^{T_B(\tau)}]) : wt$ and $T_B(D) : wt$.*

Proof. The first item follows by induction on the structure of expressions, the second by induction on the structure of processes. The third part holds, since introduction of process variables is either not changed by the transformation, or for the buffer-components it is easy to check that well-formedness is not changed. The last part follows by induction on the structure of contexts.

Corresponding typing properties hold for contexts, so that T_B forms a translation in the sense of [27]. It is easy to see that:

Lemma 4.2 (Compositionality). *The translation $T_B : \lambda^\tau(\mathsf{fchb}) \to \lambda^\tau(\mathsf{fch})$ is compositional, i.e., for all p, D we have $T_B(D)[T_B(p)] = T_B(D[p])$, and for all τ, and $e : \tau$ and $C[[\cdot]^\tau]$, we have $T_B(C)[T_B(e)] = T_B(C[e])$.*

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

We argue that the buffer implementation, described by T_B above, is correct. To this end, we will prove T_B convergence equivalent in this section, and use compositionality (Lemma 4.2). By Proposition 2.1, this entails that T_B is observationally correct and adequate.

Lemma 4.3 (T_B preserves success). *Let p be a $\lambda^\tau(\mathsf{fchb})$-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.

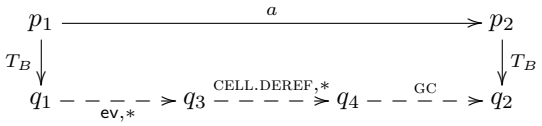
Lemma 4.4 (Context translation of T_B). *If D is a process context of $\lambda^\tau(\mathsf{fchb})$, then $T_B(D)$ is a process context. If E is an evaluation context of $\lambda^\tau(\mathsf{fchb})$, then $T_B(E)$ is an evaluation context in $\lambda^\tau(\mathsf{fch})$.*

Note that the corresponding property does not hold for future evaluation contexts. As an example consider the 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 $\lambda^\tau(\mathsf{fch})$ it is unavoidable that essentially the 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.

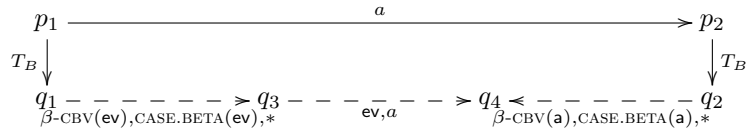
4.3 Observational Correctness and Adequacy of the Translation T_B

Proposition 4.5 ($\downarrow$-preservation of T_B). *For every $\lambda^\tau(\mathsf{fchb})$-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 4.3 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:

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

The reduction after the translation is as follows:

$$\begin{array}{l} y{\Leftarrow}put\langle x, T_B(v)\rangle) \mid (\nu x_p, x_g, x_s, x_h)(x{\Leftarrow}\langle x_p, x_g, x_s, x_h\rangle \\ \quad \mid\ (\nu h, f, h', f')(h\,\mathsf{h}\,f \mid h'\,\mathsf{h}\,f' \mid x_p\,\mathsf{c}\,\mathsf{true} \mid x_g\,\mathsf{c}\,f \mid x_s\,\mathsf{c}\,f' \mid x_h\,\mathsf{c}\,h)) \end{array}$$

This will reduce to (we show also some intermediate states)

$$\begin{array}{ll} & y{\Leftarrow}\mathsf{wait}\ \mathsf{true}; \ldots \mid (\nu h_1)(\nu f_1)h_1\,\mathsf{h}\,f_1 \mid (\nu x_p, x_g, x_s, x_h)(x{\Leftarrow}\langle x_p, x_g, x_s, x_h\rangle \\ & \quad \mid\ (\nu h, f, h', f')(h\,\mathsf{h}\,f \mid h'\,\mathsf{h}\,f' \mid x_p\,\mathsf{c}\,f_1 \mid x_g\,\mathsf{c}\,f \mid x_s\,\mathsf{c}\,f' \mid x_h\,\mathsf{c}\,h)) \\ \to & y{\Leftarrow}\mathbf{exch}(x_h, h_1) \ldots \mid (\nu h_1)(\nu f_1)h_1\,\mathsf{h}\,f_1 \mid (\nu x_p, x_g, x_s, x_h)(x{\Leftarrow}\langle x_p, x_g, x_s, x_h\rangle \\ & \quad \mid (\nu h, f, h', f')(h\,\mathsf{h}\,f \mid h'\,\mathsf{h}\,f' \mid x_p\,\mathsf{c}\,f_1 \mid x_g\,\mathsf{c}\,f \mid x_s\,\mathsf{c}\,T_B(v) \mid x_h\,\mathsf{c}\,h)) \\ \to & y{\Leftarrow}\mathsf{unit} \mid (\nu h_1)(\nu f_1)h_1\,\mathsf{h}\,f_1 \mid (\nu x_p, x_g, x_s, x_h)(x{\Leftarrow}\langle x_p, x_g, x_s, x_h\rangle \\ & \quad \mid (\nu h, f, h', f')(h\,\mathsf{h}\,\bullet \mid h'\,\mathsf{h}\,f' \mid x_p\,\mathsf{c}\,f_1 \mid x_g\,\mathsf{c}\,f \mid x_s\,\mathsf{c}\,T_B(v) \mid x_h\,\mathsf{c}\,h_1) \mid f{\Leftarrow}\mathsf{true}) \\ \xrightarrow{\text{CELL.DEREF}} & \\ & y{\Leftarrow}\mathsf{unit} \mid (\nu h_1)(\nu f_1)h_1\,\mathsf{h}\,f_1 \mid (\nu x_p, x_g, x_s, x_h)(x{\Leftarrow}\langle x_p, x_g, x_s, x_h\rangle \\ & \quad \mid (\nu h, f, h', f')(h\,\mathsf{h}\,\bullet \mid h'\,\mathsf{h}\,f' \mid x_p\,\mathsf{c}\,f_1 \mid x_g\,\mathsf{c}\,\mathsf{true} \mid x_s\,\mathsf{c}\,T_B(v) \mid x_h\,\mathsf{c}\,h_1) \mid f{\Leftarrow}\mathsf{true}) \\ \xrightarrow{\text{GC}} & y{\Leftarrow}\mathsf{unit} \mid (\nu h_1)(\nu f_1)h_1\,\mathsf{h}\,f_1 \mid (\nu x_p, x_g, x_s, x_h)(x{\Leftarrow}\langle x_p, x_g, x_s, x_h\rangle \\ & \quad \mid x_p\,\mathsf{c}\,f_1 \mid x_g\,\mathsf{c}\,\mathsf{true} \mid x_s\,\mathsf{c}\,T_B(v) \mid x_h\,\mathsf{c}\,h_1) \end{array}$$

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 *buffer*, *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.

Theorem 3.24 now shows that all non-ev-reductions in U_c are correct $\lambda^\tau(\mathsf{fch})$-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.

The other parts of the proof of convergence equivalence of T_B require a more careful analysis.

Invariants of the Buffer-Implementation. We analyze the global state of the $\lambda^\tau(\mathsf{fch})$-translation $T_B(p)$ during reduction. In this analysis we will focus on the reductions of *put* and *get*. The reduction of *buffer* provides no problems, since all reductions are correct according to Theorem 3.24.

The execution of each instance of *put* or *get* 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.) are ignored in the following analysis.

The following analysis will be used for any number of buffers, but we will first restrict the arguments 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. 9, where the indexing is according to the sequential execution. For the **put**-encoding:

  (1) $\mathsf{wait}\ (\mathbf{exch}(x_p, f))$; (1a) for **exch** . . .; (1b) for wait
  (2) $\mathbf{exch}(x_s, v_i)$;
  (3) $(\mathbf{exch}(x_h, h_{p,i}))(\mathsf{true})$ (3a) for **exch** . . .; (3b) for handle-binding

  For the **get**-encoding:

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

  For the analysis of the invariants, we 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 ignored for the moment.

- 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 **buffer** 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:

$1a$	$\ldots$ $(\mathbf{exch}(x_p, f))$;	$x_p := f_{p,i}$ where $n_p = i$
$1b$	wait $f_{p,i-1}$	synchronization
2	$\mathbf{exch}(x_s, v_i)$;	provided $h_{p,i}$ has been bound
		$x_s := v_i$ where now $k := i$
$3a$	$(\mathbf{exch}(x_h, h_{p,i}))(\mathsf{true})$	$x_h := h_{p,i}$
$3b$	$(h_{g,i-1})(\mathsf{true})$	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:

$1a$	$\ldots$ $(\mathbf{exch}(x_g, f))$;	$x_g := f_{g,i}$ 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 $k := i$
$3a$	$\ldots(\mathbf{exch}(x_h, h_{g,i}))\ldots$	$x_h := h_{g,i}$
$3b$	$h_{p,i}(\mathsf{true})$	$h_{p,i}$ will be used, $f_{p,i}\Leftarrow\mathsf{true}$ and v returned

The complete execution of *buffer* 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 *buffer*, 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 occurrences, and nowhere else. This only holds, since we look for the images of processes by the translation 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.

Referring to Fig. 9, the internal code-pointer is denoted 1a, 1b, 2, 3a, 3b. For instance, we describe subexpressions of *put* as follows: (1) for wait (**exch**(x_p, f)) with (1a) for **exch** ... and (1b) for wait, (2) for **exch**(x_s, v_i), and (3) for (**exch**$(x_h, h_{p,i})$)(true) with (3a) for **exch** ... and (3b) for handle binding. The subexpressions of *get* are described similarly.

Our analysis implies the following sequencing constraint of reductions in $\lambda^\tau(\mathsf{fch})$, where $\xrightarrow{x,a,b}$ means a reduction step for a particular buffer (implementation) x of the *put*/*get*-instance b with code-pointer a.

Lemma 4.6. *For a fixed buffer x the following sequence relations holds in any $\xrightarrow{\mathsf{ev}}$-reduction U:*
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 $\xrightarrow{x,2,b_2}$.

Proof. The analysis above shows that if for a fixed buffer x, a specific instance b of *put* or *get* performs a reduction (1b)-reduction step, then the reductions (2) (3a) (3b) of this instance can also be executed, in this sequence, and no other instance of *put* or *get* for this buffer x can make a reduction in between, which proves the first part.

This does not hold for (1a). However, the analysis of the global state above shows that (1a) corresponds to an enqueue for the access to the cell x_s, within *put*-instances (*get*-instances, respectively).

Proposition 4.7 ($\downarrow$-reflection of T_B). *For every $\lambda^\tau(\mathsf{fchb})$-process p, $T_B(p)\downarrow \Rightarrow p\downarrow$.*

Proof. Suppose that for the $\lambda^\tau(\mathsf{fchb})$-process p there is a reduction U of $T_B(p)$ to a successful process p_ω. Then the $\lambda^\tau(\mathsf{fchb})$-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 4.6.

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.

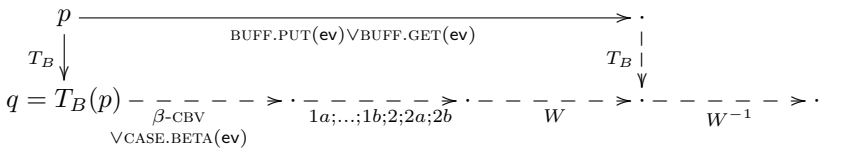
(b) The same can be done for the type 1a-reductions, which will result in a reduction, where 1a, 1b and 2-reductions of the same instance are executed without other intermediate reductions. For the 1a-type it is important to note that the sequence of $\xrightarrow{x,1a}$-reductions is the same as the type-2-reductions, if the sorting criteria is the body of the *put*, *get*s.

(c) It is also easy to see that the 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 4.6).

In a similar way we can rearrange the reduction from a *buffer*, 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 *buffer* 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 4.5: Whenever there is an a-reduction with $a \in \{\text{FUT.DEREF(ev)}, \text{LAZY.TRIGGER(ev)}\}$ 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 4.5, 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 $Q \xrightarrow{\text{FUT.DEREF(ev)},\text{LAZY.TRIGGER(ev)},*} T_B(p_\omega)$ and hence by Theorem 3.24 and Lemma 4.3, we have $p\downarrow$.

Proposition 4.8 ($\Downarrow$-reflection of T_B). *For every $\lambda^\tau(\mathsf{fchb})$-process p, $T_B(p)\Downarrow \Rightarrow p\Downarrow$.*

Proof. Suppose that for the $\lambda^\tau(\mathsf{fchb})$-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 4.5, 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 4.7 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), Theorem 3.24 is used to show $T_B(p)\uparrow$.

The proof of the following proposition is more intricate:

Proposition 4.9 ($\Downarrow$-preservation of T_B). *For all $p \in \lambda^\tau(\mathsf{fchb})$: $p\Downarrow \Rightarrow T_B(p)\Downarrow$.*

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

$$\begin{array}{ccc} p & & \\ {\scriptstyle T_B}\downarrow & & \\ q = T_B(p) & \xrightarrow[\mathsf{ev},U]{} & q_\omega\Uparrow \end{array}$$

According to the previous analysis of the intermediate states of *buffer*, *put*, *get*-operations, and using a similar construction as in the proof of Proposition 4.7, 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 *buffer*, *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 *buffer*, *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 *buffer*-reductions can be adjusted: It is easy to see that for the reductions of a single *buffer*-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 *buffer*, 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 *buffer*.
2. Now we rearrange the reduction steps of *put*, *get*, where we make implicit use of Lemma 4.6. 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 4.6 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 $q' \xrightarrow{\text{newhandled}} q_\omega$. Then Theorem 3.24 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 4.5 shows that $p_\omega\downarrow$ is impossible, hence $p_\omega\Uparrow$, which implies $p\uparrow$.

Propositions 4.5, 4.7, 4.8, 4.9, and 2.1 imply:

Theorem 4.10 (Convergence equivalence of T_B). *The translation T_B is convergence equivalent. In fact, it is observationally equivalent: for all C and e, $C[e]$ and $T_B(C)[T_B(e)]$ have the same convergence behavior.*

This directly shows the following.

Theorem 4.11 (Observational correctness of T_B). *The translation T_B is observational correct and adequate.*

For the proofs in the next section, we need also correctness of several transformations in $\lambda^\tau(\mathsf{fchb})$, which follow from adequacy of the translation T_B.

Theorem 4.12 (Correct transformations in $\lambda^\tau(\mathsf{fchb})$). *The following holds:*

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

Proof. Let $\approx$ be a program transformation mentioned in the claim. Due to adequacy of T_B by Theorem 4.10, it is sufficient to show $T_B(p_1) \sim T_B(p_2)$ for all $(p_1, p_2) \in \approx$ such that p_1, p_2 are well-typed and well-formed. As already argued (see Lemma 4.4) the translation T_B of D- and E-contexts of $\lambda^\tau(\mathsf{fchb})$ results in D and E-contexts of $\lambda^\tau(\mathsf{fch})$. The same holds for values. Thus the correctness of the reduction rules β-CBV(ev), THREAD.NEW(ev), CELL.NEW(ev), LAZY.NEW(ev) and CASE.BETA(ev) is easy to obtain by using the according equivalence for $\lambda^\tau(\mathsf{fch})$. For the rules using F-contexts (FUT.DEREF(ev) and LAZY.TRIGGER(ev)), similar as in the proof of Proposition 4.5 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 $\lambda^\tau(\mathsf{fch})$. Correctness of BUFF.NEW(ev) follows by inspecting the encoding (see proof of Proposition 4.5): The translation only consists of reductions that are correct transformations of $\lambda^\tau(\mathsf{fch})$.

We can show that the embedding $\iota_B : \lambda^\tau(\mathsf{fch}) \to \lambda^\tau(\mathsf{fchb})$ is fully abstract. However, we have to leave open the full abstractness of T_B.

Theorem 4.13 (Full abstraction of the embedding). *The embedding $\iota_B : \lambda^\tau(\mathsf{fch}) \to \lambda^\tau(\mathsf{fchb})$ is fully abstract.*

Proof. This follows from a general fully abstractness theorem of the embedding for extensions in [25].

Remark 4.14. Full abstraction cannot be shown for the translation T_B, since Proposition 2.3 is not applicable for the following subtle reason: T_B changes the types, and using a global environment Γ before the translation, we have in fact: $T_B : \lambda^\tau(\mathsf{fchb})(\Gamma) \to \lambda^\tau(\mathsf{fch})(T_B(\Gamma))$. Note also that we have types for the expressions but only one type for processes. In order to find an embedding, we would have to construct a reverse embedding $\iota : \lambda^\tau(\mathsf{fch})(T_B(\Gamma)) \to \lambda^\tau(\mathsf{fchb})(\Gamma)$, which is not possible, since then there are well-typed expressions that cannot be embedded as (well-typed) expression in $\lambda^\tau(\mathsf{fchb})$, e.g. an expressions $(\mathbf{case}\ x \ldots)$ in $\lambda^\tau(\mathsf{fch})$, where the variable x is of type $\mathsf{buf\ bool}$ in $\lambda^\tau(\mathsf{fchb})$.

4.4 Observational Correctness of the Implementation

A common approach to the specification of abstract data types, in the sequential case, is by an axiomatic description of the operations. The results developed above allow us to prove that the buffers of $\lambda^\tau(\mathsf{fchb})$ satisfy such axioms. Using adequacy of T_B (Theorem 4.11), the implied correctness of transformations for $\lambda^\tau(\mathsf{fchb})$ (Theorem 4.12), and correctness of program transformations for $\lambda^\tau(\mathsf{fch})$ (Theorem 3.24), one can show that the following rules for **put** and **get** are correct:

$$\begin{array}{ll} (\textsc{det.put}) & (\nu x).E[\mathbf{put}(x,v)] \mid x\,\mathsf{b}- \longrightarrow (\nu x).E[\mathbf{unit}] \mid x\,\mathsf{b}\,v \\ (\textsc{det.get}) & (\nu x).E[\mathbf{get}\ x] \mid x\,\mathsf{b}\,v \quad \longrightarrow (\nu x).E[v] \mid x\,\mathsf{b}- \end{array}$$

These rules are like BUFF.PUT(ev) and BUFF.GET(ev), but restricted to sequentially used buffers. Then, $\mathbf{get}(\mathbf{put}(\mathbf{buffer}\,u, v)) \sim v$ and similar equivalences follow.

More generally, we would like to show that the code in Fig. 9 correctly implements the specification of buffers, even if they are used in a non-sequential context. In other words, any use of buffers should give rise to the same observations, whether one computes with buffers abstractly using the specification, or concretely using the implementation. Informally, such a result states that the implementation as well as the specification can be considered as different realizations of an abstract data type of buffers. Since formally, the two live in different calculi, we use convergence equivalence (Theorem 4.10) of the translation T_B directly, rather than arguing by its adequacy as done for (DET.PUT) and (DET.GET) above.

More particularly, let $e : \tau'$ be any "client" making use of buffers: $e : \tau'$ is a $\lambda^\tau(\mathsf{fchb})$-program that may have free occurrences of the variables $b : \mathsf{unit} \to \mathsf{buf}\tau$, $p : \mathsf{buf}\tau \times \tau \to \mathsf{unit}$ and $g : \mathsf{buf}\tau \to \tau$ but does not otherwise contain the buffer primitives, and τ and τ' are $\lambda^\tau(\mathsf{fch})$ types. Such client programs are not affected by the encoding T_B induced by the implementation; we have $T_B(C[\,]) = C$ for the context C defined as $(\lambda\langle b, p, g\rangle.\, e)\,[\,]$. Thus, convergence equivalence (Propositions 4.5, 4.7, 4.8 and 4.9) yields

$$C[\langle \mathbf{buffer}, \lambda\langle x, y\rangle.\, \mathbf{put}(x,y), \mathbf{get}\rangle]\xi \Leftrightarrow C[\langle \mathit{buffer}, \mathit{put}, \mathit{get}\rangle]\xi$$

for all observations $\xi \in \{\downarrow, \Downarrow\}$.

$$
\begin{array}{ll}
T_H(h\,\mathsf{h}\,f) & \triangleq (\nu f')(f \overset{susp}{\Longleftarrow} \mathsf{let}\ v = \mathbf{get}\ f'\ \mathsf{in}\ \mathbf{put}(f', v); v \mid f'\,\mathsf{b}\,- \mid h \Leftarrow \lambda z.\mathbf{put}(f', z)) \\
T_H(h\,\mathsf{h}\,\bullet) & \triangleq h \overset{susp}{\Longleftarrow} h \\
T_H(\mathbf{handle}) & \triangleq \lambda x.\mathsf{let}\ f' = \mathbf{buffer\ unit} \\
 & \qquad\qquad f\ = \mathbf{lazy}\ (\lambda\_.\mathsf{let}\ v = \mathbf{get}\ f'\ \mathsf{in}\ \mathbf{put}(f', v); v) \\
 & \qquad\qquad h\ = \mathbf{thread}\ (\lambda\_.\lambda z.\mathbf{put}(f', z)) \\
 & \qquad\quad \mathsf{in}\ x\ f\ h\ \mathsf{end} \\
T_H(p) & \triangleq \text{homomorphically wrt. the term structure}
\end{array}
$$

Fig. 10. Encoding of handles using buffers

5 Handled Futures are Encodable with Buffers

In this section we show that we can encode handled futures using buffers. Let $\lambda^\tau(\mathsf{fcb})$ be the subcalculus of $\lambda^\tau(\mathsf{fchb})$ where handles are removed. More precisely, in $\lambda^\tau(\mathsf{fcb})$ the components $y\,\mathsf{h}\,x$, $\mathsf{h}\,\bullet$, and the constant **handle** are removed from the syntax of processes, expressions, evaluation contexts etc. Consequently, the reductions HANDLE.BIND(ev), and HANDLE.NEW(ev) are also dropped.

We show that there exists a translation $T_H : \lambda^\tau(\mathsf{fchb}) \to \lambda^\tau(\mathsf{fcb})$ which is fully abstract. The translation T_H is defined in Fig. 10. It does not change the types.

The idea of the translation is to simulate the synchronization effect of handles using the synchronization mechanism of one-place buffers. We consider the encoding of a handle component $y\,\mathsf{h}\,x$. An empty buffer represents the ability to bind the handled future x, i.e. binding of a handle consists in performing a put operation on the buffer. If the handled future x is accessed for the first time, then it becomes bound to the content of the filled buffer and another put-operation is performed on the buffer to ensure that the buffer remains full. The encoding of the handled future using lazy threads ensures that the possibility that a handle is not used in a successful reduction is translated into a successful reduction in the buffer implementation. Using (non-lazy) threads in the encoding would end up in a suspended get-operation, which is never successful.

The encoding of the constant **handle** generates the translated components of a handle when it is applied to an argument (modulo some BETA(ev)-reductions).

The encoding of the used handle is different, compared to the result of the reduction of an encoded handler use. Nevertheless, since a HANDLE.BIND(ev)-operation on a used handle is not possible and leads to a must-divergent process, it is sufficient to introduce the process component $h \overset{susp}{\Longleftarrow} h$, which fails as soon as an encoded HANDLE.BIND(ev)-operation is performed. Moreover, after we have proved adequacy of T_H, we can verify that $h \overset{susp}{\Longleftarrow} h \sim h\,\mathsf{h}\,\bullet$ in $\lambda^\tau(\mathsf{fchb})$ and $\lambda^\tau(\mathsf{fch})$, resp. (see Remark 5.17). The $f \overset{susp}{\Longleftarrow} \mathbf{get}\ f'$ and the **lazy**-operator in the encoding are necessary, since otherwise the future f would enforce the concurrent evaluation of $\mathbf{get}\ f'$, even if f does not occur in an E-context.

We first show some properties of T_H. The translation T_H is compatible with typing:

Lemma 5.1 (Type correctness of T_H). *Let e and p be $\lambda^\tau(\mathsf{fchb})$-expressions and processes, and C, D be $\lambda^\tau(\mathsf{fchb})$-expression and -process contexts, respectively.*

1. *If $e : \tau$ then $T_H(e) : T_H(\tau)$.*
2. *If p is well-typed, then $T_H(p)$ is well-typed.*
3. *If p is well-formed, then $T_H(p)$ is well-formed.*
4. *For $C[[\cdot]^\tau] : wt$ and $D : wt$ we have $T_H(C[\cdot]^{T_H(\tau)}]) : wt$ and $T_H(D) : wt$.*

Proof. This follows by induction of the structure of expressions, processes, and contexts, respectively. The condition on well-formedness holds: the introduction of process variables is not changed, and for the handle components the well-formedness remains intact, as can be checked easily.

Thus, T_H is a translation in the sense of [27].

Lemma 5.2 (Compositionality of T_H). *The translation $T_H : \lambda^\tau(\mathsf{fchb}) \to \lambda^\tau(\mathsf{fcb})$ is compositional, i.e., for all p, D, e, C, we have $T_H(D)[T_H(p)] = T_H(D[p])$ and $T_H(C)[T_H(e)] = T_H(C[e])$.*

Proposition 5.3. *The following properties of T_H hold:*

- *For all expressions $e \in \lambda^\tau(\mathsf{fchb})$: e is a $\lambda^\tau(\mathsf{fchb})$-value iff $T_H(e)$ is a $\lambda^\tau(\mathsf{fcb})$-value.*
- *For all processes $p \in \lambda^\tau(\mathsf{fchb})$: p is a successful $\lambda^\tau(\mathsf{fchb})$-process iff $T_H(p)$ is a successful $\lambda^\tau(\mathsf{fcb})$-process.*
- *$T_H(D)$ is a D-context for $\lambda^\tau(\mathsf{fcb})$ iff D is a D-context for $\lambda^\tau(\mathsf{fchb})$.*
- *$T_H(E)$ is a E-context for $\lambda^\tau(\mathsf{fcb})$ iff E is an E-context for $\lambda^\tau(\mathsf{fchb})$.*
- *$T_H(F)$ is a F-context for $\lambda^\tau(\mathsf{fcb})$ iff F is a F-context for $\lambda^\tau(\mathsf{fchb})$.*

Proof. The first part holds since the constant **handle** is translated into a value, and other values are translated homomorphically. Since (used) handle components are successful for $\lambda^\tau(\mathsf{fchb})$ and their translations are also successful for $\lambda^\tau(\mathsf{fcb})$, and since values are translated to values, the second part follows. The remaining parts use the first property.

Due to the last proposition the hard cases for proving convergence equivalence of T_H are the encodings of HANDLE.NEW(ev)- and HANDLE.BIND(ev)-reductions. All other reduction are inherited by the translation.

We first lift program equivalences from $\lambda^\tau(\mathsf{fchb})$ to $\lambda^\tau(\mathsf{fcb})$. Let $\iota_H : \lambda^\tau(\mathsf{fcb}) \to \lambda^\tau(\mathsf{fchb})$ be the identity translation from $\lambda^\tau(\mathsf{fcb})$ into $\lambda^\tau(\mathsf{fchb})$. Obviously, ι_H is compositional and convergence equivalent and hence it is observationally correct and adequate. In fact, at the end of the section we will prove that ι_H is fully-abstract.

Theorem 5.4 (Correct transformations in $\lambda^\tau(\mathsf{fcb})$). *The following holds:*

- *All reduction rules of $\lambda^\tau(\mathsf{fcb})$ are correct, with the exception of* CELL.EXCH(ev), BUFF.GET(ev), *and* BUFF.PUT(ev).

- *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. This follows by Theorem 4.12 and adequacy of ι_H.

Hence, we must prove some necessary properties for the encoded reductions involving handles. This requires us to prove correctness of some additional program transformations in $\lambda^\tau(\mathsf{fcb})$ in a series of lemmas.

Since T_H is compositional, for using our methods for proving observational correctness and adequacy, it is sufficient to show convergence equivalence.

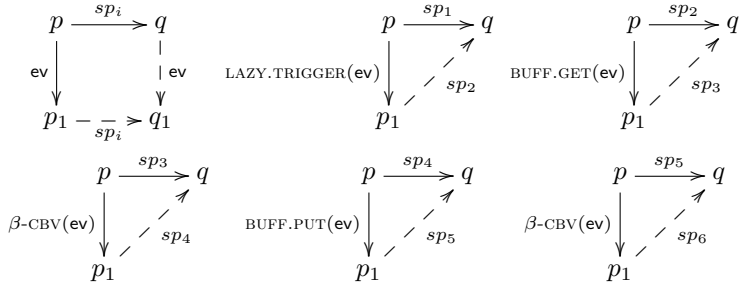
Let l_i, $i = 1, \ldots, 6$ and r be the following (meta-)processes:

$$
\begin{aligned}
l_1 &:= (\nu f')(y \overset{susp}{\Longleftarrow} \mathsf{let}\ v = \mathbf{get}\ f'\ \mathsf{in}\ \mathbf{put}(f', v); v \mid f'\,\mathsf{b}\,w \mid x \Leftarrow \lambda z.\mathbf{put}(f', z)) \\
l_2 &:= (\nu f')(y \Leftarrow \mathsf{let}\ v = \mathbf{get}\ f'\ \mathsf{in}\ \mathbf{put}(f', v); v \mid f'\,\mathsf{b}\,w \mid x \Leftarrow \lambda z.\mathbf{put}(f', z)) \\
l_3 &:= (\nu f')(y \Leftarrow \mathsf{let}\ v = w\ \mathsf{in}\ \mathbf{put}(f', v); v \mid f'\,\mathsf{b}\,- \mid x \Leftarrow \lambda z.\mathbf{put}(f', z)) \\
l_4 &:= (\nu f')(y \Leftarrow \mathbf{put}(f', w); w \mid f'\,\mathsf{b}\,- \mid x \Leftarrow \lambda z.\mathbf{put}(f', z)) \\
l_5 &:= (\nu f')(y \Leftarrow (\lambda\_.w)\,\mathbf{unit} \mid f'\,\mathsf{b}\,w \mid x \Leftarrow \lambda z.\mathbf{put}(f', z)) \\
l_6 &:= (\nu f')(y \Leftarrow w \mid f'\,\mathsf{b}\,w \mid x \Leftarrow \lambda z.\mathbf{put}(f', z)) \\
r &:= y \Leftarrow w \mid x \overset{susp}{\Longleftarrow} x
\end{aligned}
$$

Let sp be the union of the program transformations (sp_i) defined as $D[l_i] \to D[r]$ for $i = 1, \ldots, 6$.

Lemma 5.5. *For all* $p \in \lambda^\tau(\mathsf{fcb})$ *the following holds: if* $p \xrightarrow{\mathsf{ev},n} p'$ *where* p' *is successful and* $p \xrightarrow{sp_i} q$*, then* $q \xrightarrow{\mathsf{ev},\leq n} q'$ *where* q' *is successful.*

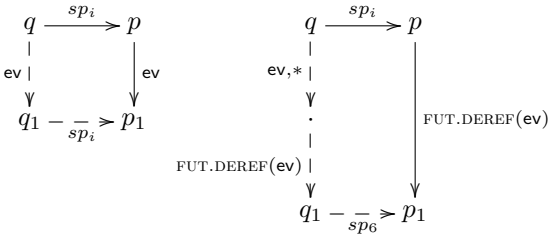
Proof. We use induction on the length n of the given reduction sequence for p. For the base case, $n = 0$, it is obvious that if p is successful, then also q is a successful process, i.e. the claim holds. For the induction step let us assume that $p \xrightarrow{\mathsf{ev}} p_1 \xrightarrow{\mathsf{ev},n-1} p_n$, where p_n is successful, and $p \xrightarrow{sp_i} q$. We now inspect the cases, which may occur, and show how the forking situation $p_1 \xleftarrow{\mathsf{ev}} p \xrightarrow{sp_i}$ can be closed, such that the induction hypothesis can be applied:



A final case is that the future x is dereferenced by the reduction $p \xrightarrow{\mathsf{ev}} p_1$. In this case p and q are always must-divergent processes, and thus these cases cannot occur. In all other cases we can apply the induction hypothesis to $p_1 \xrightarrow{sp_i} q$ or $p_1 \xrightarrow{sp_i} q_1$, which then shows the claim.

Lemma 5.6. *For all $p \in \lambda^{\tau}(\mathsf{fcb})$: if $p{\downarrow}$ and $q \xrightarrow{sp_i} p$, then $q{\downarrow}$.*

Proof. We use induction on the number n of reductions of a given reduction sequence from p to a successful process. If $n = 0$ then p is successful and q must be may-convergent. For the induction step, let $q \xrightarrow{sp_i} p \xrightarrow{\mathsf{ev}} p_1 \xrightarrow{\mathsf{ev},n-1} p_n$ where p_n is successful. Inspecting the possibilities for $q \xrightarrow{sp_i} p \xrightarrow{\mathsf{ev}} p_1$ shows that the following cases are sufficient:



A final case is that $p \xrightarrow{\mathsf{ev}} p_1$ triggers the lazy future x. Then both p and q are must-divergent processes. The diagrams show that we can apply the induction hypothesis to $q_1 \xrightarrow{sp_i} p_1$ to derive the demanded reduction sequence for q.

Lemma 5.7. *For all $p \in \lambda^{\tau}(\mathsf{fcb})$ holds: if $p \xrightarrow{\mathsf{ev},n} p'$ where p' is must-divergent and $p \xrightarrow{sp_i} q$, then $q \xrightarrow{\mathsf{ev},\leq n} q'$ where q' is must-divergent.*

Proof. The proof is analogous to the proof of Lemma 5.5, where the base case of the induction holds, since Lemma 5.6 implies that $p{\Uparrow} \Longrightarrow q{\Uparrow}$.

Proposition 5.8. *For $p_1, p_2 \in \lambda^{\tau}(\mathsf{fcb})$ with $p_1 \xrightarrow{sp} p_2$ it holds: $p_1 \sim p_2$.*

Proof. Due to Lemmas 5.5, 5.6, 5.7 it is sufficient to show that $p_1 \xrightarrow{sp} p_2$ and $p_2{\uparrow}$ implies $p_1{\uparrow}$. This proof is analogous to the proof of Lemma 5.6. The base case follows from Lemma 5.5.

Proposition 5.9 (↓-preservation of T_H). *For all $\lambda^{\tau}(\mathsf{fchb})$-processes p, the following holds: if $p{\downarrow}$, then $T_H(p){\downarrow}$.*

Proof. The proof is by induction on the length of a successfully ending reduction for p. The induction base is covered by Proposition 5.3. For the induction step let $p \xrightarrow{\mathsf{ev}} p'$. As induction hypothesis we use that $T_H(p'){\downarrow}$. Due to the properties of the context translation (Proposition 5.3) it is easy to see that all reductions of $\lambda^{\tau}(\mathsf{fchb})$ except for HANDLE.BIND(ev) and HANDLE.NEW(ev) can be transferred to the encoding in $\lambda^{\tau}(\mathsf{fcb})$, i.e. if $p \xrightarrow{a,\mathsf{ev}} p'$ with $a \notin \{\text{HANDLE.BIND}(\mathsf{ev}), \text{HANDLE.NEW}(\mathsf{ev})\}$ then $T_H(p) \xrightarrow{a,\mathsf{ev}} T_H(p')$. Hence, for these cases we have $T_H(p){\downarrow}$.

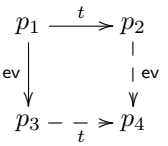
For $p \xrightarrow{\text{HANDLE.BIND}(\mathsf{ev})} p'$ we have: $T_H(p) \xrightarrow{\text{FUT.DEREF}(\mathsf{ev})} \xrightarrow{\beta\text{-CBV}(\mathsf{ev})} \xrightarrow{\text{BUFF.PUT}(\mathsf{ev})} \xrightarrow{sp_1} T_H(p')$. Since these transformations are either ev-reductions or correct , $T_H(p'){\downarrow}$ implies $T_H(p){\downarrow}$.

For $p \xrightarrow{\text{HANDLE.NEW}(\mathsf{ev})} p'$ it holds: $T_H(p) \xrightarrow{\beta\text{-CBV}(\mathsf{ev})} \xrightarrow{\text{BUFF.NEW}(\mathsf{ev})} \xrightarrow{\beta\text{-CBV}(\mathsf{ev})} \xrightarrow{\text{LAZY.NEW}(\mathsf{ev})} \xrightarrow{\beta\text{-CBV}(\mathsf{ev})} \xrightarrow{\text{THREAD.NEW}(\mathsf{ev})} \xrightarrow{\beta\text{-CBV}(\mathsf{ev})} \xrightarrow{\beta\text{-CBV}(\mathsf{ev})} \xrightarrow{\beta\text{-CBV}(\mathsf{f})} T_H(p')$ where β-CBV(f) is the following restriction of β-CBV(a): The expression context C in the transformation must be flat, i.e. the hole marker is not below a binding construct for expressions (λ-binder, or pattern in a **case**-expression). Since all these transformations are correct (Theorem 5.4), we have $T_H(p')\downarrow$ implies $T_H(p)\downarrow$.

Before we can prove reflection of may-convergence for T_H we need to show that some transformations can be commuted wrt. ev-reductions.

Lemma 5.10. *Let* $p_1, p_2 \in \lambda^\tau(\mathsf{fcb})$ *with* $p_1 \xrightarrow{t} p_2$ *where* $t \in \{\beta\text{-CBV}(\mathsf{f}), \text{BUFF.NEW}(\mathsf{ev}), \text{LAZY.NEW}(\mathsf{ev}), \text{THREAD.NEW}(\mathsf{ev}), \text{FUT.DEREF}(\mathsf{ev})\}$. *If* $p_1 \xrightarrow{\mathsf{ev},n} p_1'$ *where* p_1' *is successful (must-divergent, resp.), then there exists* p_2' *with* $p_2 \xrightarrow{\mathsf{ev},\leq n} p_2'$ *where* p_2' *is successful (must-divergent, resp.).*

Proof. We first prove the claim for reduction sequences ending in successful processes. We use induction on n. If $n = 0$, then p_1 is successful and p_2 must be successful, too. For the induction step it is easy to verify that all mentioned transformations fulfil the diagram shown on the right or that $p_2 = p_3$. For the latter case the claim obviously holds, for the former case we apply the induction hypothesis to $p_3 \xrightarrow{t} p_4$ resulting in a reduction sequence for p_4 of length $< n$. Appending this sequence to $p_2 \xrightarrow{\mathsf{ev}} p_4$ then shows the claim. Proving the claim for reduction sequences ending in must-divergent processes is analogous except for the base case of the induction: If p_1 is must-divergent, then p_2 is must-divergent, since all the mentioned transformations preserve contextual equivalence (see Theorem 5.4).

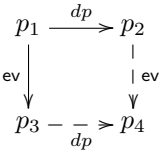


We define the program transformation $\xrightarrow{dp}$ as follows, where f' does not occur in E and w is a value.

$$D[(\nu f')(E[\mathbf{put}(f', w)] \mid y \overset{susp}{\Longleftarrow} \mathsf{let}\ v = \mathbf{get}\ f'\ \mathsf{in}\ \mathbf{put}(f', v); v \mid f'\,\mathsf{b}\,- \mid x \Leftarrow \lambda z.\mathbf{put}(f', z))]$$
$$\to D[E[\mathbf{unit}] \mid (\nu f')(y \overset{susp}{\Longleftarrow} \mathsf{let}\ v = \mathbf{get}\ f'\ \mathsf{in}\ \mathbf{put}(f', v); v \mid f'\,\mathsf{b}\,w \mid x \Leftarrow \lambda z.\mathbf{put}(f', z))]$$

Lemma 5.11. *Let* $p_1, p_2 \in \lambda^\tau(\mathsf{fcb})$ *with* $p_1 \xrightarrow{dp} p_2$. *If* $p_1 \xrightarrow{\mathsf{ev},n} p_1'$ *where* p_1' *is successful (must-divergent, resp.), then there exists* p_2' *with* $p_2 \xrightarrow{\mathsf{ev},\leq n} p_2'$ *where* p_2' *is successful (must-divergent, resp.).*

Proof. We first show the part for successfully ending reduction sequences. We use induction on n. For the base case p_1 cannot be successful, since (dp) is also an BUFF.PUT(ev)-reduction. For the induction step one can verify that the diagram shown on the right must hold or that $p_2 = p_3$. This shows that we can apply the induction hypothesis to $p_3 \xrightarrow{dp} p_4$ to derive a successful reduction sequence for p_4 of length $< n$. This sequence can be appended to $p_2 \xrightarrow{\mathsf{ev}} p_4$ resulting in the demanded reduction sequence.



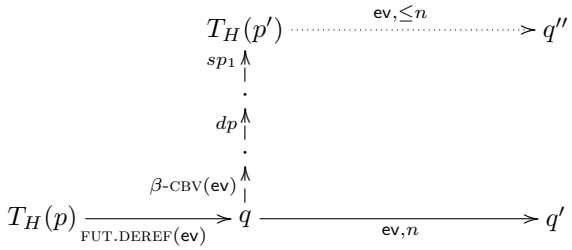
The part for reduction sequences ending in a must-divergent process is analogous where the base case of the induction holds: if $p_1 \Uparrow$ then $p_2 \Uparrow$ must hold, since (dp) is also an ev-reduction.

Proposition 5.12 ($\downarrow$-reflection of T_H). *For all $\lambda^\tau(\mathsf{fchb})$-processes p, the following holds: If $T_H(p)\downarrow$, then $p\downarrow$.*

Proof. Let $p \in \lambda^\tau(\mathsf{fchb})$ and $T_H(p)\downarrow$. We show how to derive a successful reduction sequence for p. We remember the encoded handles and handle operations in the image of T_H (e.g. by using an adequate labelling) and use induction on the length of the given reduction sequence for $T_H(p)$. If $T_H(p)$ is successful, then Proposition 5.3 implies that p must be successful, too.

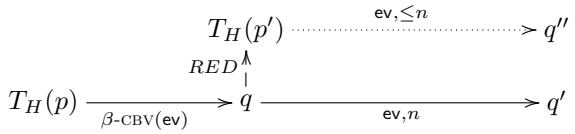
Now let $T_H(p) \xrightarrow{\mathsf{ev}} q \xrightarrow{\mathsf{ev},n} q'$ where q' is successful. There are three cases:

- The reduction $T_H(p) \xrightarrow{\mathsf{ev}} q$ is not the first reduction of an encoded HANDLE.NEW(ev) or HANDLE.BIND(ev)-reduction. For these cases it holds $p \xrightarrow{\mathsf{ev}} p'$ and $T_H(p') = q$. Applying the induction hypothesis to $T_H(p')$ shows the claim.
- The reduction $T_H(p) \xrightarrow{\mathsf{ev}} q$ is the first reduction of an encoded HANDLE.BIND(ev) reduction, i.e. it is an FUT.DEREF(ev)-reduction and $p \xrightarrow{\text{HANDLE.BIND}(\mathsf{ev})} p'$. Then we complete the encoded HANDLE.BIND(ev)-reduction as shown by the dashed arrows in the following diagram:



  It is sufficient to show that there exists q'' with $T_H(p') \xrightarrow{\mathsf{ev},\leq n} q''$ where q'' is successful, since then we can apply the induction hypothesis to p'. The existence of the reduction sequence $T_H(p') \xrightarrow{\mathsf{ev},\leq n} q''$ follows by Lemmas 5.10, 5.11, and 5.5 applied to the given reduction sequence $q \xrightarrow{\mathsf{ev},n} q'$.
- The reduction $T_H(p) \xrightarrow{\mathsf{ev}} q$ is the first reduction of an encoded HANDLE.NEW(ev) reduction, i.e. it is an β-CBV(ev)-reduction and $p \xrightarrow{\text{HANDLE.NEW}(\mathsf{ev})} p'$. Then we complete the encoded HANDLE.NEW(ev)-reduction as shown by the dashed arrows in the following diagram where

$$RED := \xrightarrow{\text{BUFF.NEW}(\mathsf{ev})} \xrightarrow{\beta\text{-CBV}(\mathsf{ev})} \xrightarrow{\text{LAZY.NEW}(\mathsf{ev})} \xrightarrow{\beta\text{-CBV}(\mathsf{ev})} \xrightarrow{\text{THREAD.NEW}(\mathsf{ev})} \xrightarrow{\beta\text{-CBV}(\mathsf{ev})} \xrightarrow{\beta\text{-CBV}(\mathsf{ev})} \xrightarrow{\beta\text{-CBV}(\mathsf{f})}$$



Applying Lemma 5.10 several times shows that there exists a reduction $T_H(p') \xrightarrow{\mathsf{ev},\leq n} q''$. Hence, we can apply the induction hypothesis to p'.

Proposition 5.13. *The translation T_H is convergence equivalent.*

Proof. Propositions 5.9 and 5.12 show that T_H preserves and reflects may-convergence. Preservation and reflection of must-convergence follows by showing that T_H preserves and reflects may-divergence. The proofs are similar, where the base cases of the inductions are valid since from preservation and reflection of may-convergence it also follows that T_H preserves and reflects must-divergence.

Since T_H is compositional and convergence equivalent, we have:

Theorem 5.14 (Observational Correctness of T_H). *The translation T_H is observational correct and adequate.*

The identity translation $\iota_H : \lambda^\tau(\mathsf{fcb}) \to \lambda^\tau(\mathsf{fchb})$ (see Theorem 5.4) is an embedding and obviously compositional and convergence equivalent, and since the translation T_H is injective on types, Proposition 2.3 is applicable:

Theorem 5.15. *The translations T_H and ι_H are fully-abstract.*

Using ι_H and ι_B (see Theorem 4.13) we can define direct translations T'_B, T'_H from $\lambda^\tau(\mathsf{fcb})$ into $\lambda^\tau(\mathsf{fch})$ and vice versa as follows: $T'_B = T_B \circ \iota_H$, $T'_H = T_H \circ \iota_B$. Since observational correctness, full-abstraction and adequacy are preserved by the composition of translation we have:

Proposition 5.16. *$T'_B : \lambda^\tau(\mathsf{fcb}) \to \lambda^\tau(\mathsf{fch})$ is observational correct and adequate and $T'_H : \lambda^\tau(\mathsf{fch}) \to \lambda^\tau(\mathsf{fcb})$ is fully abstract.*

Remark 5.17. Adequacy of T_H (T'_H, resp.) implies that used handles are equivalent to suspended black holes, i.e. $x\,\mathsf{h}\,\bullet \sim x \overset{susp}{\Longleftarrow} x$ in $\lambda^\tau(\mathsf{fchb})$ and $\lambda^\tau(\mathsf{fch})$. This follows, since the translations are syntactically equal $\lambda^\tau(\mathsf{fcb})$ processes, i.e. $T_H(x\,\mathsf{h}\,\bullet) \equiv x \overset{susp}{\Longleftarrow} x \equiv T_H(x \overset{susp}{\Longleftarrow} x)$.

6 Encoding Buffers with Cells and Busy-wait

The results obtained above lead to the question if buffers (and therefore also handled futures) can be encoded in a calculus without either synchronization primitive. In this section we partly answer this question. We encode buffers and buffer operations as cells and operations on cells by giving a translation $T_{RB} : \lambda^\tau(\mathsf{fcb}) \to \lambda^\tau(\mathsf{fc})$. The translation will be a so-called busy-wait encoding. Our semantics will show that the busy-wait encoding is observationally correct

$$
\begin{array}{l}
\mathit{buffer} \triangleq \lambda\_.\,\mathsf{let}\ x_g = \mathbf{ref}\ \mathsf{false} \\
\qquad\qquad\qquad\qquad\quad x_p = \mathbf{ref}\ \mathsf{true} \\
\qquad\qquad\qquad\qquad\quad x_s = \mathbf{ref}\ \mathit{Nothing} \\
\qquad\qquad\qquad\qquad \mathsf{in}\ \mathbf{thread}\ \lambda\_.\langle x_g, x_p, x_s\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathit{get} \triangleq\ \mathbf{thread}(\lambda\ \mathit{ig}.\,\lambda\ x.\,\mathsf{let}\ \langle x_g, x_p, x_s\rangle = x \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{getOK} = \mathbf{exch}(x_g, \mathsf{false}) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathsf{in\ if}\ \mathit{getOK} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{then\ let\ Just}\ v = \mathbf{exch}(x_s, \mathsf{Nothing}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{dummy} = \mathbf{exch}(x_p, \mathsf{true}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{in}\ v \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ \mathit{ig}\ x)
\end{array}
$$

$$
\begin{array}{l}
\mathit{put} \triangleq\ \mathbf{thread}(\lambda\ \mathit{ip}.\,\lambda\ \langle x, v\rangle.\,\mathsf{let}\ \langle x_g, x_p, x_s\rangle = x \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{putOK} = \mathbf{exch}(x_p, \mathsf{false}) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathsf{in\ if}\ \mathit{putOK} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{then}\ \mathbf{exch}(x_s, \mathsf{Just}\ v); \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{exch}(x_g, \mathsf{true}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ \mathit{ip}\ \langle x, v\rangle)
\end{array}
$$

Fig. 11. Busy-wait encoding of buffers

and adequate. However, at least from a performance point of view, these kinds of encoding should be avoided, since the knowledge where processes are waiting for a certain event gets lost.

We will use a data type $\mathit{Maybe}(a)$ with the constructors $\mathsf{Nothing}$ and unary constructor Just: $a \to \mathit{Maybe}(a)$. This simplifies the encoding since no dummy values are needed.

Fig. 11 shows the encoding of the buffer operations that induces the translation $T_{RB} : \lambda^{\tau}(\mathsf{fcb}) \to \lambda^{\tau}(\mathsf{fc})$. On buffers it is defined as follows:

$$
\begin{array}{l}
T_{RB}(x\,\mathsf{b}\,v) \triangleq (\nu x_g, x_p, x_s)(x \Leftarrow (x_g, x_p, x_s) \mid x_p\,\mathsf{c}\,\mathsf{false} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid x_g\,\mathsf{c}\,\mathsf{true} \mid x_s\,\mathsf{c}\,\mathsf{Just}\ T_{RB}(v)) \\
T_{RB}(x\,\mathsf{b}\,-) \triangleq (\nu x_g, x_p, x_s)(x \Leftarrow (x_g, x_p, x_s) \mid x_p\,\mathsf{c}\,\mathsf{true} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid x_g\,\mathsf{c}\,\mathsf{false} \mid x_s\,\mathsf{c}\,\mathsf{Nothing})
\end{array}
$$

Lemma 6.1. *The encoding T_{RB} is compositional.*

Lemma 6.2 (T_{RB} preserves success). *Let p be a $\lambda^{\tau}(\mathsf{fcb})$-process.*

1. *If p is successful, then so is $T_{RB}(p)$. In particular, $T_{RB}(p)\Downarrow$ in this case.*

2. If $T_{RB}(p)$ is successful, then p is also a successful process.

Theorem 6.3. *All of the following are correct transformations for $\lambda^\tau(\mathsf{fc})$:*

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

Proof. This follows from Theorem 3.24, since the identity transformation $id : \lambda^\tau(\mathsf{fc}) \to \lambda^\tau(\mathsf{fch})$ is compositional and convergence equivalent, and hence adequate.

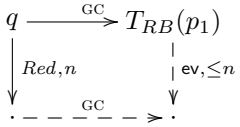
Lemma 6.4. *Let p be a $\lambda^\tau(\mathsf{fcb})$-process. If $p{\downarrow}$, then also $T_{RB}(p){\downarrow}$.*

Proof. The construction follows the same scheme as the construction of the handle-encoding of buffers. We show that for p and reduction sequence *Red* from p to a successful process there is also a reduction from $T_{RB}(p)$ to a successful process. The induction is on the length of *Red*. The base case is Lemma 6.2. Thus consider a reduction sequence $p \xrightarrow{a} p_1\ Red'$. In the standard case, the reduction can be immediately transferred, such that $T_{RB}(p) \xrightarrow{a} T_{RB}(p_1)$.
If the redex is of the form (**buffer** v) and the reduction is $p \xrightarrow{\text{BUFF.NEW(ev)}} p_1$, then the body of the encoding can be reduced and requires several reductions ending in $T_{RB}(p_1)$.
The non-standard case is that the redex is of the form (**get** x) or **put**(x, v), where x is a variable. There are several subcases:
If x is the buffer-variable, and the next reduction is an BUFF.PUT(ev) or BUFF.GET(ev), then the respective code can be executed, and we have $T_{RB}(p) \xrightarrow{*}\xrightarrow{\text{GC}} T_{RB}(p_1)$, where the final garbage collection removes the intermediately generated thread for the *get* or *put*-code, respectively. Since GC is correct according to Theorem 6.3, we can use induction. If x is not a buffer-variable, but a future, then either a dereferencing or triggering has to be done. Then $p \xrightarrow{a(\mathsf{ev})} p_1$ can be mirrored (in the dereferencing case) as follows: $T_{RB}(p) \xrightarrow{\text{THREAD.NEW}}\xrightarrow{\beta\text{-CBV}}\xrightarrow{\text{FUT.DEREF}}\xrightarrow{\beta\text{-CBV}}\xrightarrow{a(\mathsf{ev})}\xleftarrow{\beta\text{-CBV}}\xleftarrow{\text{FUT.DEREF}}\xleftarrow{\beta\text{-CBV}}\xleftarrow{\text{THREAD.NEW}} T_{RB}(p_1)$. Similarly for the triggering case. Since all reductions before and after $\xrightarrow{a(\mathsf{ev})}$ are correct transformations due to Theorem 6.3, the induction can be completed also in this case.

Lemma 6.5. *Let p be a $\lambda^\tau(\mathsf{fcb})$-process. If $T_{RB}(p){\downarrow}$, then also $p{\downarrow}$.*

Proof. We show that for p and a reduction sequence *Red* from $T_{RB}(p)$ to a successful process there is also a reduction from p to a successful process. The induction will be on the length of *Red*. However, since the reduction steps of a *get*, *put*, *buffer* may be interleaved with other reductions, several rearrangements of the reduction *Red* are necessary to enable the induction proof. The base case is that $T_{RB}(p)$ is a successful process. Then p is also a successful process.
Now let *Red* be a reduction of $T_{RB}(p)$ to a successful process. We assume that

the reduction steps that come from different instances of *put*, *get*, *buffer* can be identified in the reduction sequence *Red*. If there are busy-wait loops in the reduction, i.e. reduction of $\mathbf{exch}(x_g, \mathsf{false})$ or $\mathbf{exch}(x_p, \mathsf{false})$ with result false, then the reduction steps until the next $\mathbf{exch}(x_g, \mathsf{false})$ or $\mathbf{exch}(x_p, \mathsf{false})$, respectively of the same instance of *put*, *get* can be removed from the reduction. Thus we can assume that $\mathbf{exch}(x_g, \mathsf{false})$ or $\mathbf{exch}(x_p, \mathsf{false})$ is only reduced if the result will be true. Now we can construct the reduction of p. There are different cases:
If the first reduction step is not from an *put*, *get*, *buffer*-instance, then the reduction is also performed for p.
If the first reduction step is from *buffer*, then all the reduction steps that belong to this instance of *buffer* can be moved to the left, not changing their relative order, until they are in a contiguous sequence. Then there is some p_1, such that *Red* is exactly $T_{RB}(p) \xrightarrow{*,\mathsf{ev}} T_{RB}(p_1)$, and $p \xrightarrow{\text{BUFF.NEW}(\mathsf{ev})} p_1$.
If there is a reduction step that does not belong to an instance of *put*, *get* and that can shifted to the start of the sequence (the reduction step commutes with prefix of the reduction), then we shift it and treat it as already done above.
Now we have to rearrange the reductions corresponding to *get*, *put* in order to shift the essentially first *put*, *get* to the start of the sequence. We focus on the first reduction step $\xrightarrow{a,\mathsf{ev}}$ which is $\mathbf{exch}(x_g, \mathsf{false})$ or $\mathbf{exch}(x_p, \mathsf{false})$ in *Red*. Note that due to our assumptions above the result can only be true. Let $Red = Red_1 \cdot \xrightarrow{a,\mathsf{ev}} \cdot Red_2$. We look for the reduction steps that are in Red_1 and that belong to the same instance as $\xrightarrow{a,\mathsf{ev}}$, or are triggered by a case-reduction from the instance. There are two subcases:
The first such reduction may be a $\xrightarrow{\text{FUT.DEREF}(\mathsf{ev})}$ (or an $\xrightarrow{\text{LAZY.NEW}(\mathsf{ev})}$). Then the next reduction for p is constructed as $\xrightarrow{\text{FUT.DEREF}(\mathsf{ev})}$ and Red_1 is modified as follows: $T_{RB} \xrightarrow{\text{FUT.DEREF}} q_1 Red_{1,1} \cdot Red_{1,2}$, where $Red_{1,1}$ is constructed from Red_1 by the modifications of the $\xrightarrow{\text{FUT.DEREF}}$, i.e. the argument of *get*, *put* is modified. Note that the occurrence of x in the subexpression $(ig\ x)$, or $(ip\ x)$ will never be reduced, since we assumed that there are no busy-wait loops, hence we can ignore this occurrence. The same is done if there is the first triggered reduction is $\xrightarrow{\text{LAZY.NEW}(\mathsf{ev})}$. Now we can again assume, as above that non-instance reductions are shifted to the start, if possible.
The final case is that in Red_1, i.e. before $\xrightarrow{a,\mathsf{ev}}$, there are only the initial reductions of other instances of *put*, *get*. Now all the reductions belonging to the instance of *put*, *get* with the $\xrightarrow{a,\mathsf{ev}}$-reduction can be completely shifted to the start of Red_1. It is possible to construct the next step of the reduction for p as a $\xrightarrow{\text{BUFF.GET}(\mathsf{ev})}$, or a $\xrightarrow{\text{BUFF.PUT}(\mathsf{ev})}$-reduction, such that $p \xrightarrow{\text{BUFF.GET}(\mathsf{ev})} p_1$ and $T_{RB}(p) \xrightarrow{*,\mathsf{ev}} q \xrightarrow{\text{GC}} T_{RB}(p_1)$. This corresponds to a forking situation:



It is easy to see that the forking diagram for $\xrightarrow{\text{GC}}$ holds, hence we can use induction in this case.

Lemma 6.6. *Let p be a $\lambda^{\tau}(\mathsf{fcb})$-process. If $T_{RB}(p)\Downarrow$, then also $p\Downarrow$.*

Proof. We show that $p\uparrow$ implies that $T_{RB}(p)\uparrow$. Let us assume that $p\uparrow$, hence there is a reduction sequence *Red* of p ending in a must-divergent process. This is done by induction on the length of *Red*. The base case is proved in Lemma 6.5. Now the construction and the arguments are the same as in the proof of Lemma 6.4.

The following proof is a bit harder since we have to construct and rearrange a reduction in $\lambda^{\tau}(\mathsf{fc})$, but the arguments are as for the buffer-encoding into handles.

Lemma 6.7. *Let p be a $\lambda^{\tau}(\mathsf{fcb})$-process. If $p\Downarrow$, then also $T_{RB}(p)\Downarrow$.*

Proof. The proof shows that $T_{RB}(p)\downarrow \implies p\downarrow$ by an induction on a reduction of $T_{RB}(p)$ to a must-divergent process. The arguments are similar to those of the proof of Lemma 6.5. However, we have to add reduction steps such that the construction can be performed. Thus the induction measure cannot be the length of the reduction. An appropriate induction measure is a pair (n_1, n_2), where n_1 is the number of $\mathbf{exch}(x_g, \mathsf{false})$ or $\mathbf{exch}(x_p, \mathsf{false})$-reductions as instances of *put* and *get* which result in true. The second number is the number of reductions that do not belong to an instance of *put*, *get*, *buffer*.
The base case of the induction is that $T_{RB}(p)$ is must-divergent. Then Lemma 6.4 shows that p is also must-divergent. If the reduction has length greater than 0, then we proceed as in the proof of Lemma 6.5. The construction is almost the same, where the following additional construction steps are required: If the reductions of some instance of *put*, *get*, *buffer* are incompletely performed, then there are two cases: if the $\mathbf{exch}(x_g, \mathsf{false})$ or $\mathbf{exch}(x_p, \mathsf{false})$ is present, then the missing reductions are added to the reduction sequence. Here we use the fact, that for a must-divergent q and a reduction $q \xrightarrow{*,\mathsf{ev}} q'$, the process q' is also must-divergent. We also use the fact that these reductions can be rearranged by shifting to the left. If for an instance the reductions $\mathbf{exch}(x_g, \mathsf{false})$ or $\mathbf{exch}(x_p, \mathsf{false})$ are not present, then again there are two subcases: if a dereferencing or triggering a lazy thread is enforced by the case-redex of the instance, then the same construction and shift as in the proof of Lemma 6.5 is done. If there is no triggering by the case, then the reductions can be shifted completely to the right of the sequence. Then we use Theorem 6.3 that shows that if $q \xrightarrow{a,\mathsf{ev}} q'$ where $a \in \{\xrightarrow{\text{THREAD.NEW}(\mathsf{ev})}, \xrightarrow{\beta\text{-CBV}}, \xrightarrow{\text{FUT.DEREF}(\mathsf{ev})}\}$ and $q'\Uparrow$, then also $q\Uparrow$.

Lemmas 6.4, 6.5, 6.6 and 6.7 show the following theorem:

Theorem 6.8. *The translation $T_{RB} : \lambda^{\tau}(\mathsf{fcb}) \to \lambda^{\tau}(\mathsf{fc})$ is observationally correct and adequate.*

Using the translation $\lambda^\tau(\mathsf{fch}) \xrightarrow{T'_H} \lambda^\tau(\mathsf{fcb}) \xrightarrow{T_{RB}} \lambda^\tau(\mathsf{fc})$, we obtain as a corollary:

Corollary 6.9. *The translation $T_{RB} \circ T'_H$ is an observationally correct and adequate translation from $\lambda^\tau(\mathsf{fch})$ into $\lambda^\tau(\mathsf{fc})$.*

This shows that semantically, handles and buffers are not necessary. However, the blocking and waiting by queuing is preferable for the following reason: It is far more efficient in an abstract machine, when the machine can keep track of waiting processes and "knows" when it can resume, instead of simply looping until something changes. Another reason is that the busy waiting-translation loses the knowledge when to wait or block that was available in the higher-level implementation.
A theoretical challenge that we currently leave open is this:

> Show that there is an efficiency advantage of the handle-removing and buffer-removing translations T'_H, T'_B over the busy-wait translations.

A proof of this appears to be possible by extending our current development and arguments of the semantical theory of may- and must-convergence. Specifically, one must find an appropriate notion of corresponding reduction trees, and analyze the lengths of the corresponding reduction sequences more carefully.

6.1 Connecting $\lambda^\tau(\mathsf{fc})$ to $\lambda(\mathsf{fh})$

Let $\lambda(\mathsf{f})$ be the subcalculus of $\lambda(\mathsf{fh})$ where handles are removed, i.e. the calculus that has futures, lazy futures, and reference cells. Then in exactly the same way as we encoded $\lambda^\tau(\mathsf{fch})$ into $\lambda(\mathsf{fh})$ in Section 2 by removing types and encoding case-expressions and constructors as abstractions, we can encode $\lambda^\tau(\mathsf{fc})$ into $\lambda(\mathsf{f})$ and obtain observational correctness and adequacy of this encoding. That is. if $enc' : \lambda^\tau(\mathsf{fc}) \to \lambda(\mathsf{f})$ is the encoding $enc : \lambda^\tau(\mathsf{fch}) \to \lambda(\mathsf{fh})$ with the domain restricted to processes of $\lambda^\tau(\mathsf{fc})$, then the corollary easily follows:

Corollary 6.10. *The encoding enc' is observationally correct and adequate.*

Note that enc' is not fully abstract, see Remark 3.23 for an example that can be adapted. Furthermore, it is obvious that we can embed $\lambda(\mathsf{f})$ into $\lambda(\mathsf{fh})$ using the identity translation. This translation is compositional and clearly convergence equivalent which allows us to apply Proposition 2.1:

Corollary 6.11. *The identity translation from $\lambda(\mathsf{f})$ into $\lambda(\mathsf{fh})$ is observationally correct and adequate.*

7 Discussion and Related Work

We have proved that concurrent buffers and handled futures are equivalent synchronization primitives in the lambda calculus with futures. This result can be

seen as an extended case study, illustrating 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.

Questions of expressiveness have been addressed mainly in the pi-calculus and basic process calculi [15, 6]; 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, an ongoing research topic (e.g. [10]). However, in this context usually only (simpler) simulation properties for closed programs, rather than open program fragments, are established.

One technique for relating different primitives for *sequential* languages are proofs of representation independence, which guarantees that an invariant between two implementations (or an implementation and its specification) is preserved in all programs [20]. While recent work [1, 2, 17] has extended this method from functional to stateful higher-order languages, it is not clear to us whether it can be adapted to concurrent languages. Our Section 4.4 can be viewed as a result of this kind, but it is obtained by analysis of reduction sequences rather than by a more abstract (logical) relational proof.

Proving similar correctness and expressiveness results for other base languages is possible in principle, but requires to establish a sufficiently rich equational theory first. Here, we derived sufficiently many equivalences for our proofs via adequate translations into "smaller" core calculi. Alternatives to this approach include bisimulation methods, based on suitable labelled transition systems. This has been developed for a fragment of Concurrent ML (e.g., [7]) and for higher-order languages with ML-like general references (for instance, [9]).

An very different approach to equational correctness proofs about stateful programs is the use of Hoare-style program logics. Recently, concurrent separation logic has been used to prove properties of concurrent data structures [14].

References

1. Amal Ahmed, Derek Dreyer, and Andreas Rossberg. State-dependent representation independence. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'09)*, pages 340–353. ACM, 2009.
2. Nina Bohr and Lars Birkedal. Relational reasoning for recursive types and references. In Naoki Kobayashi, editor, *Proceedings Programming Languages and Systems (APLAS'06)*, volume 4279 of *Lecture Notes in Computer Science*, pages 79–96. Springer, 2006.
3. Arnaud Carayol, Daniel Hirschkoff, and Davide Sangiorgi. On the representation of McCarthy's amb in the pi-calculus. *Theor. Comput. Sci.*, 330(3):439–473, 2005.
4. R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.
5. Cédric Fournet, Fabrice Le Fessant, Luc Maranget, and Alan Schmitt. JoCaml: A language for concurrent distributed and mobile programming. In *Advanced Functional Programming*, volume 2638 of *Lecture Notes in Computer Science*, pages 129–158. Springer Verlag, 2002.

6. Daniele Gorla. Towards a unified approach to encodability and separation results for process calculi. In *CONCUR '08: Proceedings of the 19th international conference on Concurrency Theory*, pages 492–507, Berlin, Heidelberg, 2008. Springer-Verlag.
7. Alan Jeffrey and Julian Rathke. A theory of bisimulation for a fragment of concurrent ML with local names. *Theoretical Computer Science*, 323(1-3):1–48, 2004.
8. Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Proc. POPL'96*, pages 295–308. ACM Press, 1996.
9. Vasileios Koutavas and Mitchell Wand. Small bisimulations for reasoning about higher-order imperative programs. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'06)*, pages 141–152. ACM, 2006.
10. Xavier Leroy and Sandrine Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *J. Autom. Reasoning*, 41(1):1–31, 2008.
11. Joachim Niehren, David Sabel, Manfred Schmidt-Schauß, and Jan Schwinghammer. Observational semantics for a concurrent lambda calculus with reference cells and futures. In *23rd Conference on Mathematical Foundations of Programming Semantics*, volume 173 of *Electronical notes in theoretical computer science*, pages 313–337. Elsevier, April 2007.
12. Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A concurrent lambda calculus with futures. *Theoretical Computer Science*, 364(3):338–356, November 2006.
13. C.-H. Luke Ong. Non-determinism in a functional setting. In *LICS '93: Proceedings of the Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 275–286. IEEE Computer Society, 1993.
14. Matthew J. Parkinson, Richard Bornat, and Peter W. O'Hearn. Modular verification of a non-blocking stack. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007*, pages 297–302. ACM Press, 2007.
15. Joachim Parrow. Expressiveness of process algebras. *Electronic Notes in Theoretical Computer Science*, 209:173–186, 2008.
16. Simon Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *23rd ACM Symposium on Principles of Programming Languages*, pages 295–308, St Petersburg Beach, Florida, January 1996. ACM.
17. Uday S. Reddy and Hongseok Yang. Correctness of data representations involving heap data structures. *Science of Computer Programming*, 50(1–3):129–160, March 2004.
18. Arend Rensink and Walter Vogler. Fair testing. *Inform. and Comput.*, 205(2):125–198, 2007.
19. John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.
20. John C. Reynolds. Types, abstraction, and parametric polymorphism. In *Information Processing '83*, pages 513–523. North-Holland, Amsterdam, 1983.
21. Jon G. Riecke. Fully abstract translations between functional languages. In *18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 245–254, New York, NY, USA, 1991. ACM-Press.
22. Eike Ritter and Andrew M. Pitts. A fully abstract translation between a lambda-calculus with reference types and standard ml. In *Second International Conference on Typed Lambda Calculi and Applications*, volume 902 of *Lecture Notes in Computer Science*, pages 397–413, London, UK, 1995. Springer Verlag.

23. Andreas Rossberg, Didier Le Botlan, Guido Tack, Thorsten Brunklaus, and Gert Smolka. *Trends in Functional Programming*, volume 5, chapter Alice Through the Looking Glass, pages 79–96. Munich, Germany, February 2006.
24. David Sabel and Manfred 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.
25. Manfred Schmidt-Schauß, Joachim Niehren, Jan Schwinghammer, and David Sabel. Adequacy of compositional translations for observational semantics. Frank report 33, Inst. f. Informatik, J.W.Goethe-University, Frankfurt, 2009.
26. Manfred Schmidt-Schauß and David Sabel. On generic context lemmas for lambda calculi with sharing. Frank report 27, Inst. f. Informatik, J.W.Goethe-University, Frankfurt, 2007.
27. Manfred Schmidt-Schauß, Joachim Niehren, David Sabel, and Jan Schwinghammer. Adequacy of compositional translations for observational semantics. In *5th IFIP International Conference on Theoretical Computer Science*, volume 273, pages 521–535. Springer Verlag, 2008.