Correctly Translating Concurrency Primitives

Jan Schwinghammer
Saarland University, Saarbrücken, Germany

David Sabel
Manfred Schmidt-Schauß
Goethe-University, Frankfurt, Germany

Joachim Niehren
INRIA, Lille, France, Mostrare Project

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. Our translations map waiting on handled futures to queuing of concurrent buffers and vice versa. Correctness of translations means that they preserve and reflect the observations of may- and must-convergence. As a consequence of compositionality, they are also adequate with respect to a contextually defined notion of observational program semantics. We demonstrate that our approach to the correctness of implementations applies uniformly to the whole compilation process from high-level to low-level concurrent languages.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features – concurrent programming structures; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages – operational semantics

General Terms Theory, verification

Keywords Concurrency, lambda calculus, semantics

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 (Jones et al. 1996), channels in Concurrent ML (Reppy 1999), handled futures in Alice ML (Rossberg et al. 2006), and joins in JoCaml (Fournet et al. 2002).

Alice ML is a concurrent extension of Standard ML, adding both eager and lazy threads. The objective of the present paper is to prove precise relationships between synchronization primitives in the context of Alice ML, in particular the relation between handled futures and concurrent buffers. Our first motivation is to clarify the choice of the language primitives in the design of the Alice ML language. The second motivation is to prove the correctness of the implementation of concurrent buffers in Alice ML. We use the concurrent lambda calculus with futures (Niehren et al. 2006) to model the operational semantics of the concurrent core of Alice ML. Correctness results are expressed with respect to the observational semantics for this calculus, that is based on may- and must-convergence. The third motivation is to demonstrate the usefulness of recent results and proof techniques for observational semantics for concurrent languages.

The main contribution of this paper is a comprehensive proof that concurrent buffers and handled futures are equivalent synchronization primitives. From a semantic point of view, the designers of Alice ML can thus choose either concept as a primitive and the other as a library contribution. Both translations are compositionally and preserve may- and must-convergence, which means that they are ‘observationally correct’ and allows us to transfer program equivalences back and forth. Thus, the recently developed proof techniques based on observational semantics for concurrent languages (Schmidt-Schauß et al. 2008) have been usefully applied in an ambitious setting.

Language translations and observational semantics. We now describe our approach and results in more detail. We start with an enriched core language of Alice ML, the calculus \( \lambda^c (fch) \) which extends the calculi of Niehren et al. (2006, 2007). This is a typed call-by-value lambda with futures, polymorphic data and type constructors, concurrent threads, reference cells, and handled futures. Programs in this language consist of a collection of eager and lazy threads. When a concurrent thread is spawned, it immediately returns a future, which is a placeholder for the value computed by this thread. Other threads can proceed with this placeholder until the actual value is needed, in which case they block on the future, and they resume once the value becomes available. Besides these concurrent futures associated with threads, there are futures with an explicit handler which supports single assignment of values to futures. Handled futures, one of the two synchronization primitives we study here, are called promises in Alice ML, following the proposal of Liskov and Shrir (1988).

We use a contextual observational semantics for the concurrent lambda calculus with futures, based on operationally-defined forms of may- and must-convergence (De Nicola and Hennessy 1984; Ong 1993; Carayol et al. 2005). Our form of must-convergence is similar to the should-testing of Rensink and Vogler (2007). A common feature is that fairness of execution is mirrored in the semantic theory. This combination of may- and must-convergence properly captures the non-determinism arising in concurrent programming languages (Sabel and Schmidt-Schauß 2008; Niehren et al. 2007). Given a language \( \mathcal{C} \) we write \( \equiv_c \) for the observational semantics on programs of the language \( \mathcal{C} \), which equates all programs with equal may- and must-convergence behavior in all contexts.

For correctness of translations between languages \( T : \mathcal{C} \rightarrow \mathcal{C'} \), we use the notion of observational correctness, which for compositional translations means that all programs \( p \) and \( T(p) \) exhibit the same convergence behavior. Observational correctness implies adequacy with respect to observational semantics \( =_c \) and \( =_{c'} \) (Riecke et al. 2006).
1991; Ritter and Pitts 1995; Schmidt-Schauß et al. 2008), meaning that all program transformations of C can be soundly applied on C up to the translation T. Formally, a translation T is adequate if all (equally typed) programs with equivalent translations are equivalent, i.e., if T(p₁) = T(p₂) implies p₁ ≡ p₂. If additionally the converse holds, then T is called fully abstract. While full abstraction follows from observational correctness and some additional assumptions, it plays only a minor role for our results.

Proving observational correctness of translations. In previous work (Niehren et al. 2007) we analyzed a less expressive untyped core language, the calculus λ(fh). In comparison to the calculus λ′(fch) considered here, it lacks data constructors and case expressions which are critical for our specification of buffers. For this core language we have proved a rich set of program transformations correct with respect to contextual equivalence, using diagram-based techniques based on the operational semantics. Instead of applying this technically involved and complex mechanism again to λ′(fch), we use adequacy to lift the correctness results obtained for λ(fh) to λ′(fch), by finding suitable adequate translations.

Equipped with these results about the equational theory we focus on the correctness of implementations of buffers in λ′(fch). To obtain a specification of buffers that is sufficiently rigorous for a correctness argument we extend λ′(fch) by concurrent buffers (as e.g. used for implementing buffered channels by Jones et al. (1996)), resulting in the calculus λ′(fchb). We then present an implementation of buffers into the buffer-free calculus, such that blocking buffer operations are translated into queuing and waiting. After formalizing this implementation as a translation we show its adequacy. 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 complement our result by showing that it is also possible to go into the opposite direction and correctly implement handled futures with buffers. In this case, the specification is again the calculus λ′(fchb), but now viewed as an extension of a handle-free calculus with buffers, called λ′(fcb). We provide a translation from λ′(fchb) to λ′(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. However, the implementation that underlies this last translation can result in busy-wait situations, indicating that it is of a more low-level character than the other encodings.

For proving adequacy, full abstraction and observational correctness of the various encodings we rely on compositionality (Schmidt-Schauß and Sabel 2007; Schmidt-Schauß et al. 2008), on commutation methods in order to prove invariants of implicit queuing mechanisms that arise when implementing buffers by handled futures, as well as on equivalences for λ′(fch) that we lift from λ(fh). In turn, we inherit equations for λ′(fchb) from λ′(fcb) by the adequacy of T_B.

Summary and results. Table 1 presents the syntactic features of the various calculi which are considered in this paper.

<table>
<thead>
<tr>
<th>calculus</th>
<th>typed data constructors</th>
<th>handled futures</th>
<th>buffer primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td>λ(fh)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>λ′(fc)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>λ′(fch)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>λ′(fcb)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>λ′(fchb)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 1. Overview of the calculi considered in this paper

Our main interest lies in the last three calculi, λ′(fch), λ′(fcb), and λ′(fchb), within which we formulate the implementations of buffers and of futures. The following diagram summarizes our results. Doubly lined arrows (⇒) indicate fully abstract translations, while single lined arrows (→) indicate adequate translations.

The translation T_B describes the implementation of concurrent buffers in terms of handled futures. Conversely, the translation T_H realizes the implementation of handled futures with buffers. The translations τ_B and τ_H represent the embeddings of the smaller calculus into the extended calculus λ′(fchb) that contains both buffers and handled futures as primitives. All the translations shown in the diagram are observationally correct, and thus also preserve the convergence behavior, i.e., all these implementations are indistinguishable from the respective primitive constructs. Note that we left the exact status of T_B open: we do not know if it is fully abstract.

More explanations and full proofs can be found in the accompanying technical report (Schwinghammer et al. 2009).

2. Lambda Calculus with Futures and Constructors

This section presents the calculus λ′(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 Niehren et al. (2006) by adding data constructors with recursive polymorphic type constructors.

Type and data constructors. Our encodings require n-tuples (τ₁, . . . , τ_n) of all possible types τ₁ × . . . × τ_n. For the sake of generality and uniformity, we keep the concrete signature of data and type constructors as a parameter. Such a signature Σ = (K, D) consists of a finite ranked set of type constructors κ ∈ K and a finite ranked set of data constructors k ∈ D. We denote the arities of data and type constructors by ar(κ) ≥ 0. Polymorphic types τ over Σ have the following abstract syntax, where α belongs to a fixed infinite set of type variables:

τ ∈ PolyType ::= α | unit | ref τ | τ → τ | κ(τ₁, . . . , τ_{ar(κ)})

Monomorphic types τ ∈ Type are polymorphic types without variables. We assume a unique polymorphic type upt(k) ∈ PolyType for each data constructor k ∈ D, that has the form τ_j → . . . → τ_{ar(k)} → κ(α₁, . . . , α_{ar(κ)}) where κ ∈ K and only α₁, . . . , α_{ar(κ)} may occur as type variables in τ_j for all j =
\[\tau \in \text{Type} ::= \text{unit} | \text{ref}\,\tau | \tau \rightarrow \tau | \kappa(\tau_1, \ldots, \tau_{ar(\kappa)})\]
\[e \in \text{Const} ::= \text{unit} | \text{ref}'\,\tau | \text{thread}'\,\tau | \text{lazyc}\,\tau | \text{handle}'\,\tau\]
\[\pi \in \text{Put} ::= k^c(x_1, \ldots, x_{ar(k)})\]
\[e \in \text{Exp} ::= x | c | \lambda x.e | e_1 \cdot e_2 | \text{exch}(e_1, e_2)\]
\[\{\text{case } e \in \perp_1 \Rightarrow e_1 | \ldots | \perp_m \Rightarrow e_m \mid m > 0\}
\[v \in \text{Val} ::= x | c | \lambda x.e | k^c(v_1, \ldots, v_{ar(k)})\]
\[p \in \text{Proc} ::= p_1 | p_2 | (vx)p | \langle vx\rangle(vy)p \equiv (vx)(vy)p \equiv \langle vx\rangle(p_1) | \langle vx\rangle(p_2) \mid \text{if } x \notin \text{fv}(p_2)\]

\[\begin{align*}
\Gamma(x) &= \tau \\
\tau &\leq \alpha \rightarrow \text{ref}\,\alpha \\
\tau &\leq (\alpha \rightarrow \alpha) \rightarrow \alpha \\
\end{align*}\]
\[\begin{align*}
x : \tau &\quad \text{unit : unit} \\
\tau &\leq \alpha \rightarrow \text{ref}\,\alpha \\
\tau &\leq (\alpha \rightarrow \alpha) \rightarrow \alpha \\
\end{align*}\]
\[\begin{align*}
lazy &\rightarrow \tau \\
\end{align*}\]
\[\begin{align*}
x : \tau_1 &\quad \text{ref }\tau \rightarrow \tau \\
\end{align*}\]
\[\begin{align*}
k \in \text{D}(\kappa) &\quad \forall j \in 1 \ldots ar(k), e_j : \tau_j \\
\tau &\leq \tau_1 \rightarrow \ldots \rightarrow \tau_{ar(k)} \\
\end{align*}\]
\[\begin{align*}
\{\lambda x.e : \tau_1 \rightarrow \tau_2 | e_1 : \tau_2 \rightarrow \tau_2 \}
\end{align*}\]

\[\begin{align*}
\text{D}(\kappa) = \{k_1, \ldots, k_n\} &\quad e : \kappa(\tau'_1, \ldots, \tau'_{ar(k)}) \quad \forall i = 1 \ldots n, e_i : \tau_i \\
\Gamma(x_1) &\rightarrow \ldots \rightarrow \Gamma(x_{ar(k)}) \rightarrow \kappa(\tau'_1, \ldots, \tau'_{ar(k)}) \leq \text{upt}(k_i) \\
\end{align*}\]

\[\begin{align*}
\text{Figure 1. Types, expressions and processes of } \lambda^c(fch) \\
\end{align*}\]

\[\begin{align*}
\text{Figure 2. Structural congruence of processes} \\
\end{align*}\]

As in the pi-calculus, processes \(p\) are formed from smaller components by parallel composition \(p_1 | p_2\) and new name creation \(vx)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 \(x \in v\) associate (a memory location) \(x\) to a value \(v\). Eager concurrent threads \(x \equiv e\) will eventually bind future \(x\) to the value of expression \(e\) unless it diverges or suspends; \(x\) is called a concurrent future. Lazy threads \(x \equiv v\) are suspended computations that will start once the proper value of \(x\) is needed elsewhere; we call \(x\) a lazy future. Handle components \(y \in x\) 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 \(y \cdot x\) indicates that \(y\) is a handle that has already been used to bind its associated future. A process \(p\) introduces a variable \(x\) if \(p = p_1 \parallel p_2\) for \(p_1\) a component of the following form (for some \(e\), \(v\) and \(y\)):

\[\begin{align*}
x \in v, \text{ or } x \equiv v, \text{ or } \langle vx\rangle \equiv e, \text{ or } y \in x, \text{ or } x \cdot y, \text{ or } y \cdot x \\
\end{align*}\]

A process is well-formed if no subprocess introduces any variable more than once. For instance, neither \(x \equiv e\) nor \(\langle vx\rangle \equiv e\) nor \(\langle vx\rangle \equiv v\) is well-formed.

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 \(p : \text{put}\). 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 Niehren et al. (2006). We write \(e[v'/x]\) for the (capture-free) substitution of \(x\) by \(v'\) in \(e\). It preserves the type of \(e\) if \(e' : \Gamma(x)\) holds.

\[\begin{align*}
\text{Syntactic abbreviations. We assume that the set of type constructors contains a nullary constructor } b &\in \text{K}, \text{ with nullary data constructors true and false. For convenience, we will freely use the usual syntactic sugar such as a (non-recursive) let-binding let } x_1=1, \ldots, x_n=e_n \text{ in } e \text{ and sequencing } e_1; e_2; \text{ and also use patterns in abstractions } \lambda x.e \text{ as shorthand for } \lambda x.e \text{ for } x \neq \perp \text{ and } \lambda c.x \text{ of } \tau \Rightarrow \tau' \Rightarrow \tau \text{ etc. (where } z \text{ represents an error, and for instance can be defined by the component } z.e.z). \text{ Instead of } e \in \text{true} &\Rightarrow e_1 | \text{false } \Rightarrow e_2 \text{ we write if } e \text{ then } e_1 \text{ else } e_2, \text{ and the special case if } e \text{ then true else true is written as wait } e. \text{ The}
\end{align*}\]
\[
\begin{align*}
p_1 : \text{wt} & \quad p_2 : \text{wt} \\
p_1 | p_2 & : \text{wt} \\
\text{wt} & = t : \text{wt} \\
x : \cdot t & : \text{wt} \\
x & : \cdot t \\
x : t & : \text{wt} \\
x : \text{ref } \tau & : \text{wt} \\
x & : \text{ref } \tau \cdot t \\
x : \text{ref } \tau \cdot t & : \text{wt} \\
\end{align*}
\]

\[
\begin{align*}
p : \text{wt} & \\
(p x) : \text{wt} & \\
\text{y } h & : \text{wt} \\
\text{y } h \cdot x & : \text{wt} \\
\end{align*}
\]

**Figure 4.** Well-typed processes.

**Figure 5.** Evaluation contexts

ECs

\[
\begin{align*}
E_0 & ::= [\tau] \\
E & ::= \text{\textbf{e}} \mid \nu E \mid v E \mid \text{exch}(E, e) \mid \text{exch}(v, E) \\
& \mid k(v_1, \ldots, v_{i-1}, E, e_{i+1}, \ldots, e_n) \\
\end{align*}
\]

Future ECs

\[
\begin{align*}
F_0 & ::= [\nu] v \\
F & ::= [\text{\textbf{e}}] \text{\textbf{exch}}([\nu] v) \\
& \mid [E] \text{\textbf{case}} \left( (\pi_i \Rightarrow \nu_i) \right) \downarrow i^{1\ldots n} \\
\end{align*}
\]

Process ECs

\[
\begin{align*}
D & ::= [\text{}] p | D \mid D | p \mid (p x) D \\
\end{align*}
\]

**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 \(\beta\text{-CBV}(ev)\), \(\text{CASE} \cdot \text{BETA}(ev)\) and \(\text{FUT} \cdot \text{DEREF}(ev)\) where values with bound variables may be copied, \(\alpha\)-renaming is performed before applying the next rule.

**Closure.** Rule application is closed under structural congruence and process ECs \(D\): if \(p_1 \equiv D[p_1], p_1 \rightarrow p_2\), and \(D[p_2] \equiv p_2\) then \(p_1 \rightarrow p_2\).
x\in y \mid y\leftarrow (x, x) \text{ and } x\in y \mid y \in z \text{ are successful, while } x\in x \text{ (a black hole) and } x\in (\lambda u. \lambda v. v) \mid y\leftarrow (\lambda u. \lambda v. v) \mid x\leftarrow \text{ (a deadlocked process) are ruled out.}

While reductions preserve the type of expressions, we do not have a progress lemma since the type system does not rule out all errors. However, we can characterize these situations as follows: a well-formed and well-typed closed process can either be reduced, or it is successful, or it is not successful since it has a cycle of futures, or it is not successful since it is deadlocked. Here we use the following definitions: A \textit{waiting thread} is of the form \( x\in y; \) and a \textit{deadlocked process} is a process where every thread component is finished or waiting, and there is at least one waiting thread.

We use \( p \downarrow \) to indicate that \( p \) is may-convergent, i.e., that there is a sequence of reductions \( p \rightsquigarrow p' \) 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 \( p \uparrow \) if some reduction descendant of \( p \) is must-divergent. Thus, \( p \downarrow \Leftrightarrow \neg p \downarrow \) and \( p \downarrow \Leftrightarrow \neg p \uparrow \). For \( \xi \subseteq \{\downarrow, \uparrow, \|\} \), we define contextual approximations between processes \( p_1, p_2 \) by:

\[
p_1 \approx p_2 \Leftrightarrow \forall D: D[p_1] \Rightarrow \Rightarrow D[p_2] \\xi
\]

We write \( p_1 \approx p_2 \) if both \( p_1 \leq_1 p_2 \) and \( p_1 \leq_2 p_2 \) hold, and \( p_1 \sim p_2 \) if both \( p_1 \leq_1 p_2 \) and \( p_2 \leq p_1 \) hold. The same definitions for expressions \( e_1, e_2 \) of equal type \( \tau \) and expression contexts \( C[\xi] \) result in relations \( \leq_1, \leq_2, \approx, \sim, \approx_\tau \).

Note that the contextual equivalence \( \approx_\tau \) does not distinguish between the different kinds of error situations.

**Translations.** We recall the framework of Schmidt-Schauß et al. (2008), where an abstract calculus \( C \) consists of sets of (well-typed) processes \( p, \) contexts \( D, \) and convergence predicates \( \xi. \) The calculus \( \lambda^\ast (fch) \) 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 \( C \) and \( C' \) is \textit{equivalence correct} if \( T(p) \xi \Rightarrow \Rightarrow q \xi \) for all \( p \) and all convergence predicates \( \xi. \) The translation \( T \) is \textit{compositional if} \( T \) for all contexts \( D \) and processes \( p \) we have \( T(D[T(p)]) = T(D[p]) \). A translation \( T \) is \textit{observationally correct if} for all programs \( p \) and for all contexts \( D, \) and for all convergence predicates \( \xi, T(D[p])\xi \Rightarrow \Rightarrow T(D[p])\xi \). A translation is \textit{adequate if} \( T \) reflects operational approximation, i.e., if \( T(p_1) \leq^T T(p_2) \Rightarrow p_1 \leq C p_2 \) for all \( p_1, p_2 \). Finally, if \( T \) additionally preserves inequations, i.e., if \( T(p_1) \leq^T T(p_2) \Rightarrow p_1 \leq C p_2 \) holds for all \( p_1, p_2, \) then \( T \) is \textit{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. These notions are related:

**Proposition 2.1 (Adequacy, Schmidt-Schauß et al. 2008)).** 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 calculi considered 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 \( T(e) \sim^T e' \Rightarrow T(e) \sim e' \).

It is easy to verify that translations compose:

(FUT.DEREF(a)) \( C[x] \mid x\leftarrow v \Rightarrow C[v] \mid x\leftarrow v \)

(\( \beta \)-CBV(a)) \( C[(\lambda x.e) v] \Rightarrow C[e[v/x]] \)

(CASE.BETA(a)) \( C[\text{case } k_1 (x_1, \ldots, v_{ar(k_1)}) \text{ of } (k_2 (x_2, \ldots, v_{ar(k_2)}) \Rightarrow e_i \mid i = 1 \ldots n)] \Rightarrow C[e_i[v_1/x_1, \ldots, v_{ar(k_i)}[x_{ar(k_i)}]] \)

(CELL.DEREF) \( p \mid y \in e \Rightarrow p \mid y \in v \Rightarrow p \mid y \in v \mid x \leftarrow v \)

(GC) \( p \mid (v y_1 \ldots v y_n)p' \Rightarrow p \) if \( p' \) is successful and \( y_1, \ldots, y_n \) contain all process variables of \( p' \)

(DET.EXCH) \( (v x)(y \in E[\text{exch}] (x, v_1)) \mid x \in v_2) \Rightarrow (v x)(y \in E[\text{exch}] (x, v_1)) \mid x \in v_2) \)

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.

**Figure 7. Correct transformation rules for \( \lambda^\ast (fch) \)**

**Proposition 2.2 (Composition).** Let \( C, C', C'' \) be calculi, and \( T : C \rightarrow C', T' : C' \rightarrow C'' \) be translations. Then \( T' \circ T : C \rightarrow C'' \) 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).

We also recall a criterion for fully abstract translations, which can be used if only new primitives are added to a calculus \( C'. \) The statement of this criterion in (Schmidt-Schauß et al. 2008) contains a flaw; the following corrected version is proved in the technical report (Schmidt-Schauß et al. 2009).

**Proposition 2.3 (Full abstraction for extensions).** Let \( C, C' \) be two calculi, let \( \iota : C' \rightarrow C \) (the embedding) and \( T : C \rightarrow C' \) be compositional and convergence equivalent translations, such that \( T \iota = \text{the identity on } C'-\text{programs} \), on \( C'-\text{contexts} \), and on \( C'-\text{types} \). Then \( \iota \) 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 \( T \) 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 (Schmidt-Schauß et al. 2009). Let \( C, C' \) be calculi and \( T : C \rightarrow C' \) be an observationally correct translation. Let \( C^{\circ} := T(C) \) be the subcalculus of \( C \) consisting of the images under \( T, \) and let \( \leq T \) be the preorder defined on \( C^{\circ} \). Moreover, assume that for all \( \tau, T \) is surjective on the programs of type \( \tau \) and for every \( \tau, T \) is a surjective mapping \( T : C_{\tau_1, \tau_2} \rightarrow C_{T(\tau_1), T(\tau_2)}, \) where \( C_{\tau_1, \tau_2} \) are the contexts of type \( \tau_2 \) with hole of type \( \tau_1 \). Then, for all types \( \tau \) and programs \( p_1, p_2: p_1 \leq_T p_2 \Leftrightarrow T(p_1) \leq_T \tau(T)(p_2) T(p_2). \) That is, the translation is fully abstract as translation \( T : C \rightarrow C^{\circ}. \)

**3. Correctness of Transformations in \( \lambda^\ast (fch) \)**

We will make use of program transformations, which are called \textit{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 \( (\beta \)-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 \( \tau \) is the (uniquely determined) type of the value \( v. \) The use of the framework sketched at the end of Section 2 makes it possible, via translations, to lift process equivalences from the untyped
lambda calculus with futures (Niehren et al. 2007) to correct program transformations in $\lambda^\tau(fch)$. To illustrate the technique, we establish these correctness results by using an adequate translation from $\lambda^\tau(fch)$ into $\lambda(fh)$.

We first restate known equivalences for $\lambda(fh)$. The calculus $\lambda(fh)$ is the subcalculus of $\lambda^\tau(fch)$ without constructors, case-expressions, and typing (see Niehren et al. 2007, for details).

**Proposition 3.1.** All reductions of $\lambda(fh)$ except CELL.EXCH(ev), and the transformations $\beta$-CBV(a), PUT.DEREFS(a), CELL.DEREFS, GC and DET.EXCH (see Fig. 7) are correct for $\lambda(fh)$.

**Proof sketch.** Correctness of the transformations was established previously (Niehren et al. 2007). Although the standard reduction differs slightly from the one used here, we have shown that this makes no difference for contextual equivalence in the technical report Schmidt-Schauß et al. (2008).

---

**Removing constructors and types.** In order to lift contextual equivalences from $\lambda(fh)$ to $\lambda^\tau(fch)$, we construct a translation $T_C : \lambda^\tau(fch) \rightarrow \lambda(fh)$ that is adequate. For the adequacy of this encoding it is necessary that $\lambda^\tau(fch)$ is typed; otherwise, untyped programs that get stuck due to a dynamic type error could become must-convergent after translation. Conversely, it is not possible to restrict $\lambda(fh)$ to simple types, since the encoding of case and constructors cannot be monomorphically typed. These problems are discussed by Schmidt-Schauß et al. (2008), where illustrating examples can be found.

The main part in giving the translation is to encode case and constructors, and to show that the translation has all required properties. The encoding is a variant of the classic Church encodings. Let $K = D(\kappa)$ be the set of constructors for a specific type constructor $\kappa$ and let $\tau_1, \ldots, \tau_0$ be types. By the assumptions on the signature, $K$ is non-empty. For the definition of $T_C$ we choose an arbitrary (but from now on fixed) order of the constructors in $K$, $k_1, \ldots, k_0$ where $n \geq 1$. The two key cases of the encoding $T_C$ are the following, where $l = \alpha r(k_i)$:

$T_C(k_i(e_1,\ldots,e_i)) \triangleq$

let $x_1 := T_C(e_1), \ldots, x_i := T_C(e_i)$ in $\lambda \tau_1, \ldots, \tau_0, \tau_1 x_1 \ldots x_i$ unit $T_C(\text{case } e \in (k_i, x_1, \ldots, x_i, \alpha r(k_i)) \rightarrow e_i^{-1} n) \triangleq$

$T_C(e)(\lambda x_1, \ldots, x_i, \alpha r(k_i), \lambda \alpha r(k_i)) \triangleq$

$\ldots (\lambda \alpha r(k_n), \ldots, \alpha r(k_i), \lambda \alpha r(k_i)) \triangleq$

and it is extended homomorphically to all other cases (only the translation of reference cells requires some additional care, to ensure that the translation of the stored value is again syntactically in the form of a value). The additional unit argument to $p_i$ in the encoding of constructors achieves the correct behavior in the case of nullary constructors, with respect to call-by-value semantics. Correspondingly, in the encoding of case, the additional final abstraction in each branch leads to a uniform translation also for nullary constructors. Finally, note that types are removed by this encoding, i.e. thread $\tau$ $\triangleq$ thread, ref $\tau$ $\triangleq$ ref, lazy $\tau$ $\triangleq$ lazy, and handle $\tau$ $\triangleq$ handle.

**Proposition 3.2 (Adequacy).** The translation $T_C$ is adequate.

**Remark 3.3.** Note that $T_C$ is not fully abstract. We give an example without proof: The expressions $\lambda x.\lambda^\tau x$ and $(\lambda x.\lambda x)$ if $x$ then $x$ 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 $\lambda x.\lambda^\tau x$ (\(\lambda x\)) (\(\lambda x\)). Applying both expressions to unit will result in unit and in (unit (\(\lambda x\ unit\)) (\(\lambda x\ unit\))), respectively. The first is a value, and the second is must-divergent. In particular, this shows

**Syntactic extensions:**

$\tau \in \text{Type ::= } \text{buf} \ | \ldots$

$c \in \text{Const ::= } \text{buffer} \ | \text{get} \ | \ldots$

$e \in \text{Exp ::= } \text{put}(e_1, e_2) \ | \ldots$

$p \in \text{Proc ::= } \text{xb} \rightarrow \text{xb v} \ | \ldots$

$E ::= \text{put}(E, e) \ | \text{put}(e, E) \ | \ldots$

$F ::= \text{E[put(\{\}, v)]} \ | \text{E[get(\{\})]} \ | \ldots$

**Extensions of the type system:**

$\tau \lessdot \text{unit} \rightarrow \text{buf } \alpha$

$\tau \lessdot \text{buf } \alpha \rightarrow \alpha$

$e_1 : \text{buf } \tau \rightarrow \tau$

$e_2 : \text{unit}$

$\text{put}(e_1, e_2) : \text{unit}$

$\text{xb} : \tau \rightarrow \tau$

$\text{xb} \rightarrow \text{wb v} : \tau$

**Extensions of the reduction rules:**

(\(\text{BUF.NEW}(ev)\)) $E[\text{buffer } v] \rightarrow (\nu x)(E[x] \mid x b \rightarrow \text{unit})$

(\(\text{BUF.PUT}(ev)\)) $E[\text{put}(x, v)] \mid x b \rightarrow E[\text{unit}]$

(\(\text{BUF.GET}(ev)\)) $E[\text{get } x] \mid x b v \rightarrow E[v]$

**Figure 8.** Extensions of $\lambda^\tau(fch)$ for $\lambda^\tau(fch)$

that Proposition 2.3 is not applicable: there is no identity embedding from $\lambda(fh)$ into $\lambda^\tau(fch)$, since the former is untyped.

Correctness of program transformations for $\lambda^\tau(fch)$ follows by encoding the transformations into $\lambda(fh)$ using $T_C$ and then applying adequacy of $T_C$. This gives us the main result of this section:

**Theorem 3.4.** The reduction rules of $\lambda^\tau(fch)$, except for cell.EXCH(ev), and the transformations of Fig. 7 (note the arbitrary contexts $C$ in the first three) are correct for $\lambda^\tau(fch)$.

---

**4. Concurrent Buffers are Encodable in $\lambda^\tau(fch)$**

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

The calculus $\lambda^\tau(fch)$ is extended by new primitives for concurrent buffers. This defines the calculus $\lambda^\tau(fch)$, with the syntactic extensions shown in Fig. 8. $\lambda^\tau(fch)$ has two new components: $x b \rightarrow$ which represents an empty buffer, and $x b v$ which represents a buffer that contains the value $v$. There are new constants buffer and get to create a new buffer and get 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 the new syntax, such that exactly one expression-position, which is not restricted to values, is replaced with a typed hole marker $[\ , \ ]^\tau$.

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 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 preservation then extends to the calculus $\lambda^\tau(fch)$. Contextual preorder is defined as expected: the notion of a successful process from $\lambda^\tau(fch)$ is extended so that $\lambda^\tau(fch)$ also allows $x b \rightarrow$ and $x b v$ as components of successful processes. $\lambda^\tau(fch)$ process is well-formed if (in addition to the other process variables) no buffer variables are introduced twice.

In the remainder of this section we will show that there is an observationally correct translation $T_B : \lambda^\tau(fch) \rightarrow \lambda^\tau(fch)$ which implements buffers by handled futures. The proof of observational correctness of $T_B$ requires equivalences in $\lambda(fh)$, which have been derived in Theorem 3.4.
buffer \( \triangleq \lambda_\_ \text{let } \langle h, f \rangle = \text{newhandled}, \langle h', f' \rangle = \text{newhandled}, \text{putg} = \text{ref} (\text{true}), \text{getg} = \text{ref} (f), \text{stored} = \text{ref} (f'), \text{handler} = \text{ref} (h) \)

\[
(1) \text{ in thread } \lambda_\_ (\text{putg, getg, stored, handler}) \text{ end}
\]

\[
\begin{align*}
\text{put} & \triangleq \lambda (\langle x_p, x_g, x_s, x_h \rangle, v), \text{let } \langle h, f \rangle = \text{newhandled} \quad (1) \text{ in } \text{wait} (\text{exch} (x_p, f)); \\
& \quad \text{exch} (x_s, v); \\
& \quad (\text{exch} (x_h, h)) (\text{true}) \text{ end} \\
\text{get} & \triangleq \lambda (\langle x_p, x_g, x_s, x_h \rangle). \\
& \quad \text{let } \langle h, f \rangle = \text{newhandled}; \langle h', f' \rangle = \text{newhandled} \quad (1) \text{ in } \text{wait} (\text{exch} (x_g, f)); \\
& \quad \text{exch} (x_s, v); \\
& \quad (\text{exch} (x_h, h)) \text{ (true)} \text{ end}
\end{align*}
\]

Figure 9. Implementing the buffer operations buffer, put and get, where (1), (2), (3) indicate subexpressions for later reference.

### 4.1 Implementing Buffers Using Handled Futures

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 \( \lambda' (\text{fch}) \), in terms of reference cells and handled futures. This induces a translation from \( \lambda'(\text{fchb}) \) into \( \lambda'(\text{fch}) \).

The implementation in \( \lambda'(\text{fch}) \) 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:

\[
\text{buf } \tau \triangleq \text{ref bool } \times \text{ref bool } \times \text{ref } \tau \times \text{ref } (\text{bool } \rightarrow \text{unit})
\]

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

- the guards putg and getg contain either a handled future or true (perhaps reachable via dereferencing futures),
- at most one of putg or getg contains true,
- if getg contains true then the value in stored is the current value 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 (putg, getg, stored, handler) 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 \( \lambda' (\text{fchb}) \) into \( \lambda' (\text{fch}) \): 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:

\[
\begin{align*}
T_B (x \ b \ -) & \triangleq (\nu h)(\nu f)(\nu x)(\nu s)(\nu c)(x, s, c \in \{ x_p, x_g, x_s, x_h \}) \\
& \mid (\nu h) (f) (h' f') (h f) \mid h f' \mid x_p c \ \text{true} \\
T_B (x \ b \ v) & \triangleq (\nu h)(\nu f)(\nu x)(\nu s)(\nu c)(x, s, c \in \{ x_p, x_g, x_s, x_h \}) \\
& \mid (\nu h) (f) (h f) \mid x_p c \ \text{true} \\
& \text{where } T_B (\text{buf } \tau) \triangleq \text{buf } (T_B (\tau)), \text{ and proceeding homomorphically in all other cases. Note that } T_B \text{ is not injective on the types, since buffer types are mapped to the type of 4-tuples.}
\end{align*}
\]

Formally, these replacements extend homomorphically to a mapping \( T_B : \lambda' (\text{fchb}) \rightarrow \lambda' (\text{fch}) \) on all \( \lambda' (\text{fchb}) \)-expressions, processes, and -contexts. A corresponding type translation is defined inductively, by \( T_B (\text{buf } \tau) \triangleq \text{buf } (T_B (\tau)) \), and proceeding homomorphically in all other cases. Note that \( T_B \) is not injective on the types, since buffer types are mapped to the type of 4-tuples. These mappings are compatible with typing:

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 \( C[[\tau]] : \text{wt} \) and \( D : \text{wt} \), we have \( T_B (C[[\tau]]_{T_B (\tau)}) \) : \( \text{wt} \) and \( T_B (D) : \text{wt} \).

Corresponding typing properties also hold for contexts, so that \( T_B \) forms a translation in the sense of Section 2 that is compositional:

**Lemma 4.1** (Compositional). The translation \( T_B : \lambda' (\text{fchb}) \rightarrow \lambda' (\text{fch}) \) is compositional, i.e., for all \( p \) and \( D \) we have \( T_B (D)[T_B (p)] = T_B (D[p]) \), and for all \( \tau, \alpha e : \tau \) and \( C[[\tau]] \), 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, respectively.

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 4.1). By Proposition 2.1, this entails the observational correctness of \( T_B \).

**Lemma 4.2** (\( T_B \) preserves success). Let \( p \) be a \( \lambda' (\text{fchb}) \)-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 4.3. If $D$ is a process context of $\lambda^T$ (fchb), then $T_B(D)$ is a process context. If $E$ is an evaluation context of $\lambda^T$ (fchb), then $T_B(E)$ is an evaluation context in $\lambda^T$ (fchb).

Note that a corresponding property does not hold for future evaluation contexts. As an example, consider the process $x \leftarrow \text{put}(y, x) \mid y \rightarrow \text{get} x \mid \ldots$. Assume that \text{get} is executed first, then \text{put}. For the corresponding reduction sequence in $\lambda^T$ (fchb), it is unavoidable that essentially the same sequence is used on the implementation \text{get} and \text{put}. However, the initial reductions of \text{put} may be executed earlier. (In the case of $y \leftarrow \text{get} x$, this is even enforced.) For the reduction in the implementation this means that the reduction steps of instances of \text{get} and \text{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 of the Translation $T_B$

Proposition 4.4 (↓-preservation of $T_B$). For every $\lambda^T$ (fchb)-process $p$, $p \downarrow \Rightarrow T_B(p) \downarrow$.

Proof sketch. The main idea of the proof is to rearrange a reduction sequence $U$ of $T_B(p)$ using Lemma 4.5 until the reduction steps that belong to an instance of put/get are in a contiguous block. Since future evaluation contexts do not correspond before and after the translation, we also have to use the equivalences from Theorem 3.4 for further rearrangement.

Proposition 4.6 (↓-reflection of $T_B$). For every $\lambda^T$ (fchb)-process $p$, $T_B(p) \downarrow \Rightarrow p \downarrow$.

Proof. Suppose that for the $\lambda^T$ (fchb)-process $p$ we have $p \downarrow$. We show $T_B(p) \downarrow$. Since $p \downarrow$ there is a reduction $R$ from $p$ to a process $p_0 \uparrow$. Analogous to the proof of Proposition 4.4, 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 4.6 applied to $p_0$ shows that $T_B(p_0) \downarrow$ is impossible, hence $T_B(p_0) \downarrow$ holds. By induction on the length of $R'$ (which consists of $\text{ev}$-reductions and correct transformations), Theorem 3.4 is used to show $T_B(p) \downarrow$.

The proof of the following proposition is more intricate:

Proposition 4.7 (↓-preservation of $T_B$). For all $p \in \lambda^T$ (fchb): $p \downarrow \Rightarrow T_B(p \downarrow) \downarrow$.

Proof. The detailed proof is in (Schwinghammer et al. 2009); here we give a sketch. We prove the equivalent claim that for every $\lambda^T$ (fchb)-process $p$, $T_B(p \downarrow) \downarrow \Rightarrow p \downarrow$. As in the proof of Proposition 4.6, a given reduction $R$ corresponding to $T_B(p \downarrow)$ will be rearranged and modified in order to construct a $\lambda^T$ (fchb)-reduction of $p$ that shows $p \downarrow$. This allows a similar rearrangement into 1, 2, 3-blocks and intermediate correct transformations. However, this is not possible for all instances of \text{put} and \text{get}, since some of these may be started but never completed in the reduction.

Variants of the following argument can be used to overcome this difficulty. Suppose a certain instance $m$ of \text{put} has been started within $R$, but the next reduction step, say from (3a), is missing in $R$. Let $p_m$ be the last process in $R$, for which we necessarily have $p_m \uparrow$. The commutation properties that show $3a, m \rightarrow 3b, m$ is a reduction possibility of $p_m$, i.e., $p_m \rightarrow^{3a, m} p_{\downarrow} m$, which immediately implies $p_m \downarrow \Rightarrow q_m \downarrow$ and $q_m \downarrow \Rightarrow R \downarrow$. This procedure is repeated until all partially executed instances are completed; as an invariant, the number of started instances of \text{put} and \text{get} is not increased. Now one can construct a reduction sequence showing $p \downarrow$. Finally, there is some process $p_n$ with $p_n \rightarrow p_n$ and $T_B(p_n) \downarrow = q_n$, where $q_n$ is the final process of the rearranged and extended sequence $R'$. The property $q_n \downarrow$ can be shown using Theorem 3.4. Lemma 4.4 shows that $p_n \downarrow$ is impossible, hence $p_n \uparrow$. Thus, $p \downarrow$.

Propositions 4.4, 4.6, 4.7, 4.8, and 2.1 imply:
Theorem 4.9 (Observational correctness of $T_B$). The translation $T_B$ is convergence equivalent. In fact, it is observationally correct: for all $C$ and $e$, $C[e]$ and $T_B(C)[T_B(e)]$ have the same convergence behavior.

As a consequence of this theorem, $T_B$ is also adequate. For the proofs in the next section, we need also the correctness of several transformations in $\lambda^f$ (fchb), which follow from adequacy.

Proposition 4.10 (Correct transformations in $\lambda^f$ (fchb)). All reduction rules of $\lambda^f$ (fchb) are correct, with the exception of cellul.exch(ev), buff.get(ev), and buff.put(ev). The transformations $\beta$-cbv(a), fut.deref(a), cell.deref, gc and det.exch (see Fig. 7) lifted to $\lambda^f$ (fchb) are correct.

Proof sketch. The statement follows by translating the transformations into the calculus $\lambda^f$ (fchb) using $T_B$. Then, the adequacy of $T_B$, some reasoning using Lemma 4.3, and Proposition 4.4 show the claim.

Although we do not make use of this fact, we can show that the embedding $\iota_B : \lambda^f$ (fchb) $\to \lambda^f$ (fchb) is fully abstract, using Proposition 2.3.

Corollary 4.11 (Full abstraction of the embedding). The embedding $\iota_B : \lambda^f$ (fchb) $\to \lambda^f$ (fchb) is fully abstract.

As an aside, note that we cannot use Proposition 2.3 to also show full abstraction of the translation $T_B$: the proposition requires injectivity on types, which $T_B$ does not satisfy.

4.4 Applications of Observational Correctness

We can use the observational correctness of the translation $T_B$ to derive a number of consequences for the implementation of buffers, and in turn obtain justification for considering this notion.

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 $\lambda^f$ (fchb) satisfy a number of such axioms. Using adequacy of $T_B$, the implied correctness of transformations for $\lambda^f$ (fchb) (Theorem 4.10), and correctness of program transformations for $\lambda^f$ (fchb) (Theorem 3.4), one can show that the following transformations for

$\text{put}$ and $\text{get}$ are correct:

$$(\text{det.put})(x, \text{put}(x, v)) \to \text{put}(x, v)$$

These rules are like buff.put(ev) and buff.get(ev), but restricted to sequentially used buffers. Then, $\text{get}(\text{buffer}(u, v)) \sim v$ 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 three live in different calculi, we use convergence equivalence (Theorem 4.9) of the translation $T_B$ directly, rather than arguing by its adequacy as done for (det.put) and (det.get) above.

Specifically, let $e : \tau'$ be any “client” making use of buffers: $e : \tau'$ is a $\lambda^f$ (fchb)-program that may have free occurrences of the variables $b : \text{unit} \to \text{buf}\tau, p : \text{buf}\tau \times \tau \to \text{unit}$ and $g : \text{buf}\tau \to \tau$ but does not otherwise contain the buffer primitives, and $\tau$ and $\tau'$ are $\lambda^f$ (fchb) 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 let $(b, p, g) = []$ in $e$, so convergence equivalence yields $C[(\text{buffer}, \lambda(x, y).\text{put}(x, y).\text{get})] \xi \equiv C[(\text{buffer}, \text{put}, \text{get})] \xi$

for all observations $\xi \in \{1, \bot\}$.

5. Handled Futures are Encodable with Buffers

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

We show that there exists a translation $T_H : \lambda^f$ (fchb) $\to \lambda^f$ (fchb) which is observationally correct and 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 $y \times h \times x$. 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. If (non-lazy) threads were used in the encoding instead, one would end up with a suspended get-operation which is never successful.

The encoding of the constant $\text{handle}$ generates the translated components of a handle when it is applied to an argument (modulo some $\text{beta}(ev)$-reductions). Note that, equivalently, we could have defined $h = \lambda z.\text{put}(f', z)$ for the let-binding of $h$, since $\lambda z.\text{put}(f', z) \sim \text{thread}(\lambda v.\lambda z.\text{put}(f', z))$ can be proved in $\lambda^f$ (fchb), using the correct transformations of Lemma 5.3. The variant in Fig. 10 simplifies some of the arguments below.

The encoding of the used handle is different, compared to the result of the reduction of an encoded handler use. This appears necessary since for the translation of $h \bullet$ we do not know the name $f'$ of the buffer that was previously used to translate $h \bullet f$. However, because a $\text{HANDLE}.\text{bind}(ev)$-operation on a used handle is not possible and leads to a must-divergent process, it is sufficient to introduce the process component $h \equiv h \bullet f$ for the let-binding of $h$, which fails as soon as an encoded $\text{HANDLE}.\text{bind}(ev)$-operation is performed. Moreover, after we have proved adequacy of $T_H$, we can verify that $h \equiv h \bullet h \bullet$ in $\lambda^f$ (fchb) and $\lambda^f$ (fchb), respectively (see Remark 5.9). The $f \equiv f \bullet$ and the $\text{lazy}$-operator in the encoding are necessary, otherwise the future $f$ would force the
concurrent evaluation of \( f' \) even if \( f \) does not occur in an \( E \)-context.

The translation \( T_H \) is compatible with typing, i.e. if \( p \) is well-typed and well-formed, then \( T_H(p) \) is well-typed and well-formed. Corresponding typing properties hold for contexts, so that \( T_H \) is a translation in the sense of Section 2.

**Lemma 5.1 (Compositionality of \( T_H \)).** The translation \( T_H : \Lambda^c(fchb) \to \Lambda^c(fchb) \) is compositional, i.e., for all \( p, D, e, C \), we have \( T_H(D[I_H(p)] = T_H(D[p]) \) and \( T_H(C)[T_H(e)] = T_H(C[e]) \).

We sketch the proof of observational correctness and adequacy. The following properties of \( T_H \) are easy to verify:

**Proposition 5.2.** An expression \( e \in \Lambda^c(fchb) \) is a \( \Lambda^c(fchb) \)-value if \( T_H(e) \) is a \( \Lambda^c(fchb) \)-value. A process \( p \in \Lambda^c(fchb) \) is a successful \( \Lambda^c(fchb) \)-process iff \( T_H(p) \) is a successful \( \Lambda^c(fchb) \)-process. For contexts of \( \Lambda^c(fchb) \): \( T_H(D) \) is a context, \( T_H(F) \), resp. is a D-context (\( E \)-context, \( F \)-context, resp.) for \( \Lambda^c(fchb) \) \( D \) (\( E \), \( F \), resp.) is a D-context (\( E \)-context, \( F \)-context, resp.) for \( \Lambda^c(fchb) \).

Because of this proposition, the hard cases for proving convergence equivalence of \( T_H \) are the encodings of \( \text{HANDLE}.\text{NEW}(\text{ev}) \)- and \( \text{HANDLE}.\text{BIND}(\text{ev}) \)-reductions; all other reduction are inherited by the translation. We first lift program equivalences from \( \Lambda^c(fchb) \) to \( \Lambda^c(fchb) \). Let \( \iota_H : \Lambda^c(fchb) \to \Lambda^c(fchb) \) be the identity translation from \( \Lambda^c(fchb) \) into \( \Lambda^c(fchb) \). Obviously, \( \iota_H \) is compositional and convergence equivalent, hence it is adequate. (In fact, we will prove that \( \iota_H \) is fully-abstarct at the end of this section.) As an immediate consequence of Proposition 4.10 and the adequacy of \( \iota_H \) we obtain the following correct transformations.

**Lemma 5.3 (Correct transformations in \( \Lambda^c(fchb) \)).** All reduction rules of \( \Lambda^c(fchb) \) are correct, with the exception of \( \text{CELL}.\text{EXCH}(\text{ev}) \), \( \text{BUFF}.\text{GET}(\text{ev}) \), and \( \text{BUFF}.\text{PUT}(\text{ev}) \). The transformations \( \beta-\text{CBV}(\text{a}), \text{FUT}.\text{DEREF}(\text{ev}), \text{CELL}.\text{DEREF}, \text{GC} \) and \( \text{DET}.\text{EXCH} \) (see Fig. 7) lifted to \( \Lambda^c(fchb) \) are correct.

**Proposition 5.4 (\( \downarrow \)-preservation of \( T_H \)).** For all \( \Lambda^c(fchb) \)-processes \( p \), the following holds: if \( p \) then \( T_H(p) \).

**Proof.** The proof is by induction on the length of a successfully ending reduction for \( p \). The induction base is covered by Proposition 5.2. For the induction step let \( p \xrightarrow{\downarrow} p' \). As induction hypothesis we use that \( T_H(p') \). Due to the properties of the context translation (Proposition 5.2) it is easy to see that all reductions of \( \Lambda^c(fchb) \) except for \( \text{HANDLE}.\text{BIND}(\text{ev}) \) and \( \text{HANDLE}.\text{NEW}(\text{ev}) \) can be transferred to the encoding in \( \Lambda^c(fchb) \), i.e. if \( p \xrightarrow{\downarrow} p' \) with \( \not\in \{ \text{HANDLE}.\text{BIND}(\text{ev}), \text{HANDLE}.\text{NEW}(\text{ev}) \} \) then \( T_H(p) \xrightarrow{\downarrow} T_H(p') \). Hence, for these cases we have \( T_H(p) \).

For \( p \xrightarrow{\downarrow} p' \) we have:

\[
T_H(p) \xrightarrow{\downarrow} T_H(p')
\]

Since these transformations are either ev-reductions or correct (correctness of \( sp_1 \) is proved in (Schwinghammer et al. 2009)), \( T_H(p') \) implies \( T_H(p) \).

**Proposition 5.5 (\( \downarrow \)-reflection of \( T_H \)).** For all \( \Lambda^c(fchb) \)-processes \( p \), the following holds: if \( T_H(p) \), then \( p \).

**Proof.** Let \( p \in \Lambda^c(fchb) \) and \( T_H(p) \). We show how to derive a successful reduction sequence for \( p \). We track the encoded handles and handle operations in the image of \( T_H \) (e.g. by using an appropriate labeling) and use induction on the length of the given reduction sequence for \( T_H(p) \). If \( T_H(p) \) is successful, then Proposition 5.5 implies that \( p \) must be successful, too. For the induction step let \( T_H(p) \xrightarrow{\downarrow} q' \) where \( q' \) is successful. There are three cases:

- The reduction \( T_H(p) \xrightarrow{\downarrow} q \) is not the first reduction of an encoded \( \text{HANDLE}.\text{NEW}(\text{ev}) \) or \( \text{HANDLE}.\text{BIND}(\text{ev}) \)-reduction.
  
  For these cases it holds \( p \xrightarrow{\downarrow} p' \) and \( T_H(p') = q \).

- The reduction \( T_H(p) \xrightarrow{\downarrow} q \) is the first reduction of an encoded \( \text{HANDLE}.\text{BIND}(\text{ev}) \)-reduction, i.e. it is a \( \text{FUT}.\text{DEREF}(\text{ev}) \)-reduction and \( p \xrightarrow{\downarrow} p' \). In (Schwinghammer et al. 2009) using Lemma 5.3 we show that there exists a reduction sequence \( T_H(p') \xrightarrow{\downarrow} q'' \) where \( q'' \) is successful.

- The reduction \( T_H(p) \xrightarrow{\downarrow} q \) is the first reduction of an encoded \( \text{HANDLE}.\text{NEW}(\text{ev}) \) reduction, i.e. it is a \( \beta-\text{CBV}(\text{ev}) \)-reduction and \( p \xrightarrow{\downarrow} p' \). Then again it is possible to construct a reduction sequence \( T_H(p') \xrightarrow{\downarrow} q'' \) where \( q'' \) is successful.

In all cases we can apply the induction hypothesis to \( T_H(p') \).

**Theorem 5.7 (Observational correctness and adequacy of \( T_H \)).** The translation \( T_H \) is observationally correct and adequate.

Since the identity translation \( \iota_H : \Lambda^c(fchb) \to \Lambda^c(fchb) \) which we introduced above is an embedding and obviously compositional and convergence equivalent, and since the translation \( T_H \) is injective on types, Proposition 2.3 is applicable and shows:

**Corollary 5.8 (Full abstraction).** The translations \( T_H \) and \( \iota_H \) are fully abstract.

Using \( \iota_H \) and \( \iota_B \) (see Theorem 4.11) we can define direct translations \( T_B \) and \( T'_B \) from \( \Lambda^c(fchb) \) into \( \Lambda^c(fchb) \) and vice versa, by setting \( T_B = T_H \circ \iota_H \) and \( T'_B = T_H \circ \iota_B \). Since full abstraction and adequacy are preserved under composition, we obtain an adequate translation \( T_B : \Lambda^c(fchb) \to \Lambda^c(fchb) \) and a fully abstract translation \( T'_B : \Lambda^c(fchb) \to \Lambda^c(fchb) \).

**Remark 5.9.** Adequacy of \( T_B \) (resp. \( T'_B \)) implies that used handles are equivalent to suspended black holes, i.e. \( x \bullet \sim x \xrightarrow{\text{sup}} x \) in \( \Lambda^c(fchb) \) and \( \Lambda^c(fchb) \). This follows, since the translations are syntactically equal \( \Lambda^c(fchb) \) processes, i.e. \( T_H(x \bullet \text{ev}) \equiv x \xrightarrow{\text{sup}} x \equiv T_H(x) \).
Theorem 6.1

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 as cells and buffer operations as operations on cells, by giving a translation $T_{RB} : \lambda' \to \lambda'$. Here, the target calculus $\lambda'$ is like $\lambda'$ but without buffers. The translation will be a busy-wait encoding, and our semantics will show that this is an adequate encoding. 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 $\text{option}(\tau)$ with the nullary constructor $\text{None} : \text{option}(\tau)$ and unary constructor $\text{Some} : \tau \to \text{option}(\tau)$. This simplifies the encoding since no dummy values are needed. Fig. 11 shows the encoding of the buffer operations that induces the translation $T_{RB} : \lambda' \to \lambda'$. On buffers it is defined as follows:

$$T_{RB}(x b v) \triangleq (\nu x_q, x_p, x_s)(x \leftarrow (x_q, x_p, x_s) \mid x_p \equiv \text{false} \mid x_q \equiv \text{true} \mid x_s \equiv \text{Some } T_{RB}(v))$$

$$T_{RB}(x b -) \triangleq (\nu x_q, x_p, x_s)(x \leftarrow (x_q, x_p, x_s) \mid x_p \equiv \text{true} \mid x_q \equiv \text{false} \mid x_s \equiv \text{None})$$

This encoding is compositional and preserves success, i.e., $p$ is successful iff $T_{RB}(p)$ is. Moreover, Theorem 3.4 and adequacy of the identity transformation $\text{id} : \lambda' \to \lambda'$ show that the reduction rules of $\lambda'$, except for $\text{CELL.EXCH}(\text{ev})$ and the handle-rules, and the transformations of Fig. 7, are also correct transformations for $\lambda'$. This allows us to prove convergence equivalence of $T_{RB}(p)$, which implies correctness:

**Theorem 6.1** (Observational correctness and adequacy). The translation $T_{RB} : \lambda' \to \lambda'$ is observationally correct and adequate.

As a corollary, using the translation $T_{HB} : \lambda' \to \lambda'$ described at the end of the previous section, we obtain an observationally correct and adequate translation $T_{RB} \circ T_{HB}$ from $\lambda'$ into $\lambda'$. Essentially, these translations show that handled futures and buffers are not necessary from a semantic viewpoint. However, the blocking and waiting by queuing is preferable to the busy-wait implementation given in this section, since a machine can keep track of suspended processes and notify them after changes. Accordingly, we obtain the following theoretical challenge, that we currently leave open:

Show that there is an efficiency advantage of the handle-removing and buffer-removing translations $T_{HB}, T_{RB}$ over the busy-wait translations.

A proof of this challenge could 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.

Finally, to complete the diagram from the introduction, it remains to relate $\lambda' (\text{fc})$ to the calculus $\lambda (\text{fh})$. We achieve this in two steps: First, the data constructors and case-expressions of $\lambda' (\text{fc})$ are encoded in an (untyped) intermediate calculus that only has concurrent and lazy futures and reference cells. In a second step, this intermediate calculus is embedded into the calculus $\lambda (\text{fh})$ (which additionally features handled futures). The translations used in both steps are observationally correct and adequate, hence this also holds for their composition. The details of this construction can be found in the technical report (Schwinghammer et al. 2009).

### 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 as cells and buffer operations as operations on cells, by giving a translation $T_{RB} : \lambda' (\text{fc}) \to \lambda' (\text{fc})$. Here, the target calculus $\lambda'$ is like $\lambda'$ but without buffers. The translation will be a busy-wait encoding, and our semantics will show that this is an adequate encoding. 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 $\text{option}(\tau)$ with the nullary constructor $\text{None} : \text{option}(\tau)$ and unary constructor $\text{Some} : \tau \to \text{option}(\tau)$. This simplifies the encoding since no dummy values are needed. Fig. 11 shows the encoding of the buffer operations that induces the translation $T_{RB} : \lambda' (\text{fc}) \to \lambda' (\text{fc})$. On buffers it is defined as follows:

$$T_{RB}(x b v) \triangleq (\nu x_q, x_p, x_s)(x \leftarrow (x_q, x_p, x_s) \mid x_p \equiv \text{false} \mid x_q \equiv \text{true} \mid x_s \equiv \text{Some } T_{RB}(v))$$

$$T_{RB}(x b -) \triangleq (\nu x_q, x_p, x_s)(x \leftarrow (x_q, x_p, x_s) \mid x_p \equiv \text{true} \mid x_q \equiv \text{false} \mid x_s \equiv \text{None})$$

This encoding is compositional and preserves success, i.e., $p$ is successful iff $T_{RB}(p)$ is. Moreover, Theorem 3.4 and adequacy of the identity transformation $\text{id} : \lambda' (\text{fc}) \to \lambda' (\text{fc})$ show that the reduction rules of $\lambda'$ (except for $\text{CELL.EXCH}(\text{ev})$) and the handle-rules, and the transformations of Fig. 7, are also correct transformations for $\lambda' (\text{fc})$. This allows us to prove convergence equivalence of $T_{RB}(p)$, which implies correctness:

**Theorem 6.1** (Observational correctness and adequacy). The translation $T_{RB} : \lambda' (\text{fc}) \to \lambda' (\text{fc})$ is observationally correct and adequate.

As a corollary, using the translation $T_{HB} : \lambda' (\text{fc}) \to \lambda' (\text{fc})$ described at the end of the previous section, we obtain an observationally correct and adequate translation $T_{RB} \circ T_{HB}$ from $\lambda'$ into $\lambda'$. Essentially, these translations show that handled futures and buffers are not necessary from a semantic viewpoint. However, the blocking and waiting by queuing is preferable to the busy-wait implementation given in this section, since a machine can keep track of suspended processes and notify them after changes. Accordingly, we obtain the following theoretical challenge, that we currently leave open:

Show that there is an efficiency advantage of the handle-removing and buffer-removing translations $T_{HB}, T_{RB}$ over the busy-wait translations.

A proof of this challenge could 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.

Finally, to complete the diagram from the introduction, it remains to relate $\lambda' (\text{fc})$ to the calculus $\lambda (\text{fh})$. We achieve this in two steps: First, the data constructors and case-expressions of $\lambda' (\text{fc})$ are encoded in an (untyped) intermediate calculus that only has concurrent and lazy futures and reference cells. In a second step, this intermediate calculus is embedded into the calculus $\lambda (\text{fh})$ (which additionally features handled futures). The translations used in both steps are observationally correct and adequate, hence this also holds for their composition. The details of this construction can be found in the technical report (Schwinghammer et al. 2009).

### 7. Discussion and Related Work

In this paper we have proposed a method for specifying and reasoning about implementations, based on semantics-preserving translations. We have proved that concurrent buffers and handled futures are encoded as synchronization primitives in the lambda calculus with futures, in the sense that each can correctly encode the other. This result can be seen as an extended case study of our method, and illustrates how recent proof techniques based on observational semantics permit to prove the equivalence of various concurrency primitives of realistic concurrent programming languages.

To complete the picture in our particular setting, there are two immediate open questions: Can we make precise the intuition that the translations $T_{HB}, T_{RB}$ that encode blocking by waiting and queuing and vice versa, improve upon the busy-wait encoding? And, is the translation $T_{RB}$ fully abstract?

Questions of expressiveness have been addressed mainly in the pi-calculus and basic process calculi (Parrow 2008; Gorla 2008); we are not aware of previous work on formally relating synchronization primitives in concurrent high-level languages with respect to contextual semantics. Similar issues, concerning properties of translations, arise in the verification of compilers, an ongoing research topic (e.g. Leroy and Blazy 2008). 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 (Reynolds 1983). While recent work (Ahmed et al. 2009; Bohr and Birklad 2006; Reddy and Yang 2004) has extended this method from functional to stateful higher-order languages, it is not clear to us whether it can also be adapted to concurrent languages. Our Section 4.4 can be viewed as a result of this kind, which is obtained by a detailed analysis of reduction sequences.

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 may be bisimulation methods, based on suitable labelled transition systems. For instance, bisimulation methods have been developed for an untyped, stateful higher-order language by Koutavas and Wand (2006), and for fragments of Concurrent ML (e.g., by Jeffrey and Rathke 2004). However, it is open in which way bisimilarity can characterize contextual semantics with respect to may- and must-convergence, in particular for languages like the ones considered in this paper.
References


