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 compositional 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 $\lambda^\tau(\mathsf{fch})$ which extends the calculi of Niehren et al. (2006, 2007). 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. 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 Shrira (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 $=_\mathcal{C}$ 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 $T : \mathcal{C} \to \mathcal{C}'$, 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 $=_\mathcal{C}$ and $=_{\mathcal{C}'}$ (Riecke

ML'09, August 30, 2009, Edinburgh, Scotland, UK.

1991; Ritter and Pitts 1995; Schmidt-Schauß et al. 2008), meaning that all program transformations of $\mathcal{C}'$ can be soundly applied on $\mathcal{C}$ up to the translation T. Formally, a translation T is adequate if all (equally typed) programs with equivalent translations are equivalent, i.e., if $T(p_1) =_{\mathcal{C}'} T(p_2)$ implies $p_1 =_{\mathcal{C}} p_2$. If additionally the converse holds, then T is called fully abstract. 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 $\lambda(\mathsf{fh})$. In comparison to the calculus $\lambda^\tau(\mathsf{fch})$ 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 $\lambda^\tau(\mathsf{fch})$, we use adequacy to lift the correctness results obtained for $\lambda(\mathsf{fh})$ to $\lambda^\tau(\mathsf{fch})$, by finding suitable adequate translations.

Equipped with these results about the equational theory 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 by Jones et al. (1996)), resulting in the calculus $\lambda^\tau(\mathsf{fchb})$. 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 $\lambda^\tau(\mathsf{fchb})$, 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. 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 $\lambda^\tau(\mathsf{fch})$ that we lift from $\lambda(\mathsf{fh})$. In turn, we inherit equations for $\lambda^\tau(\mathsf{fchb})$ from $\lambda^\tau(\mathsf{fch})$ 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.

calculus	typed	data constructors	handled futures	buffer primitives
$\lambda(\mathsf{fh})$	×	×	✓	×
$\lambda^\tau(\mathsf{fc})$	✓	✓	×	×
$\lambda^\tau(\mathsf{fch})$	✓	✓	✓	×
$\lambda^\tau(\mathsf{fcb})$	✓	✓	×	✓
$\lambda^\tau(\mathsf{fchb})$	✓	✓	✓	✓

Table 1. Overview of the calculi considered in this paper

Our main interest lies in the last three calculi, $\lambda^\tau(\mathsf{fch})$, $\lambda^\tau(\mathsf{fcb})$, and $\lambda^\tau(\mathsf{fchb})$, within which we formulate the implementations of buffers and of futures. The following diagram summarizes our results. Doubly lined arrows ($\Longrightarrow$) indicate fully abstract translations, while single lined arrows ($\longrightarrow$) indicate adequate translations.

$$\begin{tikzcd}
 & \lambda^\tau(\mathsf{fch}) \arrow[r, "T_C"] \arrow[dl, Rightarrow, "\iota_B"] & \lambda(\mathsf{fh}) \\
\lambda^\tau(\mathsf{fchb}) \arrow[ur, bend left, "T_B"] \arrow[dr, Rightarrow, bend left, "T_H"] & & \\
 & \lambda^\tau(\mathsf{fcb}) \arrow[r, "T_{RB}"] \arrow[ul, "\iota_H"] & \lambda^\tau(\mathsf{fc})
\end{tikzcd}$$

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 ι_B and ι_H represent the embeddings of the smaller calculi into the extended calculus $\lambda^\tau(\mathsf{fchb})$ 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.

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 between $\lambda^\tau(\mathsf{fcb})$ into $\lambda^\tau(\mathsf{fch})$.

Outline. The next section presents the calculus $\lambda^\tau(\mathsf{fch})$ underlying Alice ML, and also abstracts from its observational semantics a more general framework with respect to which we can define language translations and establish some of their basic properties. Section 3 develops the equational theory of $\lambda^\tau(\mathsf{fch})$, by showing the adequacy of translation to $\lambda(\mathsf{fh})$. This paves the way for the correctness proof for the implementation of buffers, given in Section 4. Section 5 considers the implementation of handled futures by buffers (T_H in the diagram above), and Section 6 considers an implementation of buffers in a calculus without handled futures (T_{RB} above), based on busy waiting. Open questions and related work are discussed in Section 7.

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 $\lambda^\tau(\mathsf{fch})$ 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 Niehren et al. (2006) by adding data constructors with recursive polymorphic type constructors.

Type and data 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 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 $j =$

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

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

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

Figure 2. Structural congruence of processes

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

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

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

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

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

Figure 3. Types of expressions

$1, \ldots, ar(k)$. 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 and typing 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. Thus, 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 $fv(e)$ ($fv(p)$ 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 $\mathbf{lazy}^\tau$ 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.

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 Niehren et al. (2006). 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 $\mathtt{true}$ and $\mathtt{false}$. 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 $\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$, and the special case $\mathsf{if}\ e\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ \mathsf{true}$ is written as $\mathsf{wait}\ e$. The

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

$$\frac{p{:}wt}{(\nu x)p{:}wt} \qquad \frac{}{y\,\mathsf{h}\,\bullet : wt} \qquad \frac{x{:}\tau \qquad y{:}\tau \to \mathsf{unit}}{y\,\mathsf{h}\,x : wt}$$

Figure 4. Well-typed processes.

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

Figure 5. Evaluation contexts

symbol '_' stands for an arbitrary fresh variable. Finally, we write $\mathsf{newhandled}$ as shorthand for **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 $p : wt$, 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 $\xrightarrow{*}$ ($\xrightarrow{+}$, resp.) for the reflexive-transitive (transitive, resp.) closure of $\to$. We sometimes label reductions with their name, e.g. $\xrightarrow{\text{BETA}(\mathsf{ev})}$, and $p \xrightarrow{a \vee b} q$ means that either $p \xrightarrow{a} q$ or $p \xrightarrow{b} q$.

Rule (CELL.NEW(ev)) creates new cells $z\,\mathsf{c}\,v$ with contents v. The exchange operation $\textbf{exch}(z, v_1)$ 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 preservation holds for reduction, i.e., reduction preserves well-typedness (and well-formedness) of processes.

Reduction rules.

(β-CBV(ev))	$E[(\lambda x.e)\,v] \to E[e[v/x]]$
(CELL.NEW(ev))	$E[\textbf{ref}\ v] \to (\nu z)(E[z] \mid z\,\mathsf{c}\,v)$
(CELL.EXCH(ev))	$E[\textbf{exch}(z, v_1)] \mid z\,\mathsf{c}\,v_2 \to E[v_2] \mid z\,\mathsf{c}\,v_1$
(THREAD.NEW(ev))	$E[\textbf{thread}\ v] \to (\nu z)(E[z] \mid z{\Leftarrow}v\,z)$
(FUT.DEREF(ev))	$F[x] \mid x{\Leftarrow}v \to F[v] \mid x{\Leftarrow}v$
(LAZY.NEW(ev))	$E[\textbf{lazy}\ v] \to (\nu z)(E[z] \mid z \overset{susp}{\Longleftarrow} v\,z)$
(LAZY.TRIGGER(ev))	$F[x] \mid x \overset{susp}{\Longleftarrow} e \to F[x] \mid x{\Leftarrow}e$
(HANDLE.NEW(ev))	$E[\textbf{handle}\ v] \to (\nu z)(\nu z')(E[v\,z\,z'] \mid z'\,\mathsf{h}\,z)$
(HANDLE.BIND(ev))	$E[x\,v] \mid x\,\mathsf{h}\,y \to E[\textbf{unit}] \mid y{\Leftarrow}v \mid x\,\mathsf{h}\,\bullet$

(CASE.BETA(ev))

$$E[\textbf{case}\ k_j(v_1, \ldots, v_{ar(k_j)})\ \textbf{of}\ (k_i(x_1, \ldots, x_{ar(k_i)}) \Rightarrow e_i)^{i=1\ldots n}] \to E[e_j[v_1/x_1, \ldots, v_{ar(k_j)}/x_{ar(k_j)}]]$$

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 $p_1 \to p_2$.

Figure 6. Small-step reduction relation $\to$ ($\xrightarrow{\mathsf{ev}}$, resp.) of $\lambda^\tau(\mathsf{fch})$

As an example, let $r{:}(\mathsf{ref}\ \mathsf{List}(\tau))$ be a reference cell containing τ-lists. The effect of reducing process $r\,\mathsf{c}\,v \mid (\nu x)(x{\Leftarrow}\textbf{thread}\ \lambda y.(\textbf{exch}(r, \mathsf{cons}(v', y))))$ 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 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: $p \equiv F[\textbf{case}\ x\ \textbf{of}\ \mathsf{Ack} \Rightarrow e]$. The thread may resume the computation 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 $F[e] \mid F'[\textbf{unit}] \mid x{\Leftarrow}\mathsf{Ack} \mid h\,\mathsf{h}\,\bullet$, 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.

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 *waiting thread* is of the form $x \Leftarrow \widetilde{F}[y]$ and no reduction applies; a finished thread is of the form $x \Leftarrow v$; and a *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 $p \rightarrow^* p'$ 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 $p\uparrow \Leftrightarrow \neg p\Downarrow$. For $\xi \in \{\downarrow, \Downarrow\}$, we define contextual approximations between processes p_1, 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 $\sim_\tau$.

Note that the contextual equivalence $\sim$ 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 $\mathcal{C}$ 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 programs p and for all contexts D, and for all convergence predicates ξ: $T(D[p])\xi \Leftrightarrow T(D)[T(p)])\xi$. 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 $T(p_1) \leq^{\mathcal{C}'} T(p_2) \Leftrightarrow p_1 \leq^{\mathcal{C}} p_2$ holds for all p_1, p_2, 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. 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 $T(e) \sim_{T(\tau)} T(e')$ implies $e \sim_\tau e'$.

It is easy to verify that translations compose:

(FUT.DEREF(a)) $C[x] \mid x \Leftarrow v \to C[v] \mid x \Leftarrow v$

(β-CBV(a)) $C[(\lambda x.e)\ v] \ \to\ C[e[v/x]]$

(CASE.BETA(a))
$C[\mathbf{case}\ k_j(v_1,\ldots,v_{ar(k_j)})\ \mathbf{of}\ (k_i(x_1,\ldots,x_{ar(k_i)}) \Rightarrow e_i)^{i=1\ldots n}]$
$\to C[e_j[v_1/x_1,\ldots,v_{ar(k_j)}/x_{ar(k_j)}]]$

(CELL.DEREF) $p \mid y\,\mathsf{c}\,x \mid x \Leftarrow v \ \to\ p \mid y\,\mathsf{c}\,v \mid x \Leftarrow v$

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

(DET.EXCH) $(\nu x)(y \Leftarrow \widetilde{E}[\mathbf{exch}(x, v_1)] \mid x\,\mathsf{c}\,v_2)$
$\to\ (\nu x)(y \Leftarrow \widetilde{E}[v_2] \mid x\,\mathsf{c}\,v_1)$

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 $\lambda^\tau(\mathsf{fch})$

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).*

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 (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 $\mathcal{C}, \mathcal{C}'$ 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 $\mathcal{C}'$-contexts, 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 (Schmidt-Schauß et al. 2009). 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 $\lambda^\tau(\mathsf{fch})$

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, via translations, to lift process equivalences from the untyped lambda calculus with futures (Niehren et al. 2007) to correct program transformations in $\lambda^\tau(\mathsf{fch})$. To illustrate the technique, we establish these correctness results by using an adequate translation from $\lambda^\tau(\mathsf{fch})$ into $\lambda(\mathsf{fh})$.

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

Proposition 3.1. *All reductions of* $\lambda(\mathsf{fh})$ *except* CELL.EXCH(ev), *and the transformations* β-CBV(a), FUT.DEREF(a), CELL.DEREF, GC *and* DET.EXCH *(see Fig. 7) are correct for* $\lambda(\mathsf{fh})$.

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 $\lambda(\mathsf{fh})$ to $\lambda^\tau(\mathsf{fch})$, we construct a translation $T_C : \lambda^\tau(\mathsf{fch}) \to \lambda(\mathsf{fh})$ that is adequate. For the adequacy of this encoding it is necessary that $\lambda^\tau(\mathsf{fch})$ 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 $\lambda(\mathsf{fh})$ 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 $K = \mathcal{D}(\kappa)$ 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. For the definition of T_C we choose an arbitrary (but from now on fixed) order of the constructors in K, $k_1, \ldots, k_n$ where $n \geq 1$. The two key cases of the encoding T_C are the following, where $l = ar(k_i)$:

$$T_C(k_i(e_1,\ldots,e_l)) \triangleq \\ \mathsf{let}\ x_1{=}T_C(e_1),\ldots,x_l{=}T_C(e_l)\ \mathsf{in}\ \lambda p_1,\ldots,p_n.p_i\, x_1 \ldots x_l\ \textbf{unit}$$

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

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. $\textbf{thread}^\tau \triangleq \textbf{thread}$, $\textbf{ref}^\tau \triangleq \textbf{ref}$, $\textbf{lazy}^\tau \triangleq \textbf{lazy}$, and $\textbf{handle}^\tau \triangleq \textbf{handle}$.

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* $\lambda x^{\mathsf{bool}}.x$ *and* $(\lambda x^{\mathsf{bool}}.\mathit{if}\ x\ \mathit{then}\ x\ \mathit{else}\ x)$ *are equivalent, but the translated expressions are not equivalent since they behave differently when applied to* **unit**. *The second expression is translated into* $\lambda x.x\ (\lambda\_.x)\ (\lambda\_.x)$. *Applying both expressions to* **unit** *will result in* **unit** *and in* (**unit** $(\lambda\_.\textbf{unit})$ $(\lambda\_.\textbf{unit})$)*, respectively. The first is a value, and the second is must-divergent. In particular, this 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.*

Syntactic extensions:

$$\begin{aligned} \tau \in \mathit{Type} &::= \mathsf{buf}\ \tau \mid \ldots \\ c \in \mathit{Const} &::= \textbf{buffer}^\tau \mid \textbf{get}^\tau \mid \ldots \\ e \in \mathit{Exp} &::= \textbf{put}(e_1, e_2) \mid \ldots \\ p \in \mathit{Proc} &::= x\,\mathsf{b}\,- \mid x\,\mathsf{b}\,v \mid \ldots \\ \widetilde{E} &::= \textbf{put}(\widetilde{E}, e) \mid \textbf{put}(v, \widetilde{E}) \mid \ldots \\ \widetilde{F} &::= \widetilde{E}[\textbf{put}([\,], v)] \mid \widetilde{E}[\textbf{get}\,[\,]] \mid \ldots \end{aligned}$$

Extensions of the type system:

$$\frac{\tau \preceq \mathsf{unit} \to \mathsf{buf}\ \alpha}{\textbf{buffer}^\tau : \tau} \quad \frac{\tau \preceq \mathsf{buf}\ \alpha \to \alpha}{\textbf{get}^\tau : \tau} \quad \frac{e_1 : \mathsf{buf}\ \tau \quad e_2 : \tau}{\textbf{put}(e_1, e_2) : \mathsf{unit}}$$

$$\frac{}{x\,\mathsf{b}\,- : wt} \quad \frac{x{:}\mathsf{buf}\ \tau \quad v : \tau}{x\,\mathsf{b}\,v{:}wt}$$

Extensions of the reduction rules:

(BUFF.NEW(ev))	$E[\textbf{buffer}\ v]$	$\to$	$(\nu x)(E[x] \mid x\,\mathsf{b}\,-)$ fresh x
(BUFF.PUT(ev))	$E[\textbf{put}(x, v)] \mid x\,\mathsf{b}\,-$	$\to$	$E[\textbf{unit}] \mid x\,\mathsf{b}\,v$
(BUFF.GET(ev))	$E[\textbf{get}\ x] \mid x\,\mathsf{b}\,v$	$\to$	$E[v] \mid x\,\mathsf{b}\,-$

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

Correctness of program transformations for $\lambda^\tau(\mathsf{fch})$ follows by encoding the transformations into $\lambda(\mathsf{fh})$ 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* $\lambda^\tau(\mathsf{fch})$*, except for* CELL.EXCH(ev)*, and the transformations of Fig. 7 (note the arbitrary contexts* C *in the first three) are correct for* $\lambda^\tau(\mathsf{fch})$.

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 $\textbf{buffer}^\tau$ to create a new buffer and $\textbf{get}^\tau$ 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 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 $\textbf{put}(x, v)$ 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 $\lambda^\tau(\mathsf{fchb})$. 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.

In the remainder of this section we will show that there is an observationally correct translation $T_B : \lambda^\tau(\mathsf{fchb}) \to \lambda^\tau(\mathsf{fch})$ which implements buffers by handled futures. The proof of observational correctness of T_B requires equivalences in $\lambda^\tau(\mathsf{fch})$, which have been derived in Theorem 3.4.

$$
\begin{array}{ll}
 & \textit{buffer} \triangleq \lambda\_.\ \mathsf{let}\ \langle h,f\rangle = \mathsf{newhandled}, \langle h',f'\rangle = \mathsf{newhandled},\\
 & \qquad \textit{putg} = \mathbf{ref}(\mathsf{true}), \textit{getg} = \mathbf{ref}(f),\\
 & \qquad \textit{stored} = \mathbf{ref}(f'), \textit{handler} = \mathbf{ref}(h)\\
(1) & \quad \mathsf{in}\ \mathbf{thread}\ \lambda\_.\langle \textit{putg}, \textit{getg}, \textit{stored}, \textit{handler}\rangle\ \mathsf{end}\\
\\
 & \textit{put} \triangleq \lambda\langle\langle x_p, x_g, x_s, x_h\rangle, v\rangle.\ \mathsf{let}\ \langle h,f\rangle = \mathsf{newhandled}\\
(1) & \quad \mathsf{in}\ \mathsf{wait}\ (\mathbf{exch}(x_p, f));\\
(2) & \qquad \mathbf{exch}(x_s, v);\\
(3) & \qquad (\mathbf{exch}(x_h, h))(\mathsf{true})\quad \mathsf{end}\\
\\
 & \textit{get} \triangleq \lambda\langle x_p, x_g, x_s, x_h\rangle.\\
 & \quad \mathsf{let}\ \langle h,f\rangle = \mathsf{newhandled}; \langle h',f'\rangle = \mathsf{newhandled}\\
(1) & \quad \mathsf{in}\ \mathsf{wait}\ (\mathbf{exch}(x_g, f));\\
(2) & \qquad \mathsf{let}\ v = (\mathbf{exch}(x_s, f'))\\
(3) & \qquad \mathsf{in}\ (\mathbf{exch}(x_h, h))(\mathsf{true});\ v\ \mathsf{end}\\
 & \quad \mathsf{end}
\end{array}
$$

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 $\lambda^\tau(\mathsf{fch})$, in terms of reference cells and handled futures. 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:

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

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

- 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 $\langle \textit{putg}, \textit{getg}, \textit{stored}, \textit{handler}\rangle$ 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:

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

Formally, these replacements extend homomorphically 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 $T_B(\mathsf{buf}\ \tau) \triangleq \textit{buf}\ (T_B(\tau))$, 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 $C[[\cdot]^\tau] : \textit{wt}$ and $D : \textit{wt}$ we have $T_B(C[[\cdot]^{T_B(\tau)}]) : \textit{wt}$ and $T_B(D) : \textit{wt}$.

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 (Compositionality)**.** *The translation $T_B : \lambda^\tau(\mathsf{fchb}) \to \lambda^\tau(\mathsf{fch})$ is compositional, i.e., for all p and D we have $T_B(D)[T_B(p)] = T_B(D[p])$, and for all τ, all $e : \tau$ 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, 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 $\lambda^\tau(\mathsf{fchb})$-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 4.3. *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 a 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 of the Translation T_B

Proposition 4.4 ($\downarrow$-preservation of T_B)**.** *For every $\lambda^\tau(\mathsf{fchb})$-process p, $p\downarrow \Rightarrow T_B(p)\downarrow$.*

Proof sketch. The proof can be done by induction on the length of a reduction U of p, where one has to take into account that the future evaluation contexts are not preserved, and so the sequence of reductions is not straightforward. The idea is to construct a sequence of reductions and transformations in $\lambda^\tau(\mathsf{fch})$, and then use Theorem 3.4 that shows that we have only inserted correct $\lambda^\tau(\mathsf{fch})$-transformations. □

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

Invariants of the Buffer-Implementation. We sketch an analysis of 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 in $\lambda^\tau(\mathsf{fch})$ according to Theorem 3.4.

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.

It is possible to analyze exactly the global state and the changes that belong to one buffer x. We will only informally describe the results here. The analysis has to take into account all 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. Therefore, we require the code-pointer that 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

Informally, there are two queues: one queue of *put*-instances blocking on guards that need to be activated by the respective handlers by the previous *get*, and another queue of *get*-instances that wait for activation of their guards by handlers of 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. 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 4.5. *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 $\xrightarrow{x,2,b_2}$.

Proposition 4.6 ($\downarrow$-reflection of T_B)**.** *For every $\lambda^\tau(\mathsf{fchb})$-process p, $T_B(p)\downarrow \Rightarrow p\downarrow$.*

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 4.7 ($\Downarrow$-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 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 4.4, 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 $T_B(p_0)\Uparrow$ holds. By induction on the length of R' (which consists of ev-reductions and correct transformations), Theorem 3.4 is used to show $T_B(p)\uparrow$. □

The proof of the following proposition is more intricate:

Proposition 4.8 ($\Downarrow$-preservation of T_B)**.** *For all $p \in \lambda^\tau(\mathsf{fchb})$: $p\Downarrow \Rightarrow T_B(p)\Downarrow$.*

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

Variants of the following argument can be used to overcome this difficulty. Suppose a certain instance m of *put* has been started within R, but the next reduction step, say from (3a), is missing in R. Let p_ω be the last process in R, for which we necessarily have $p_\omega\Uparrow$. The commutation properties show that $\xrightarrow{3a,m}$ is a reduction possibility of p_ω, i.e. $p_\omega \xrightarrow{3a,m} p_{\omega,1}$, which immediately implies $p_{\omega,1}\Uparrow$. Thus we extend R to $R\cdot \xrightarrow{3a,m}$. This procedure is repeated until all partially executed instances are completed; as an invariant, the number of started instances of *put*, *get* is not increased. Now one can construct a reduction sequence showing $p\uparrow$. Finally, there is some process p_ω with $p \xrightarrow{*} p_\omega$ and $T_B(p_\omega) = q_\omega$, where q_ω is the final process of the rearranged and extended sequence R'. The property $q_\omega\Uparrow$ can be shown using Theorem 3.4. Lemma 4.4 shows that $p_\omega\downarrow$ is impossible, hence $p_\omega\Uparrow$. Thus, $p\uparrow$. □

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 $\lambda^\tau(\mathsf{fchb})$, which follow from adequacy.

Proposition 4.10 (Correct transformations in $\lambda^\tau(\mathsf{fchb})$)**.** *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 sketch. The statement follows by translating the transformations into the calculus $\lambda^\tau(\mathsf{fch})$ 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 $\iota_B : \lambda^\tau(\mathsf{fch}) \to \lambda^\tau(\mathsf{fchb})$ is fully abstract, using Proposition 2.3.

Corollary 4.11 (Full abstraction of the embedding)**.** *The embedding $\iota_B : \lambda^\tau(\mathsf{fch}) \to \lambda^\tau(\mathsf{fchb})$ 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 $\lambda^\tau(\mathsf{fchb})$ satisfy a number of such axioms. Using adequacy of T_B, the implied correctness of transformations for $\lambda^\tau(\mathsf{fchb})$ (Theorem 4.10), and correctness of program transformations for $\lambda^\tau(\mathsf{fch})$ (Theorem 3.4), one can show that the following transformations for **put** and **get** are correct:

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

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 4.9) 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 $\lambda^\tau(\mathsf{fchb})$-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 $\mathsf{let}\ \langle b,p,g\rangle = [\,]\ \mathsf{in}\ e$, so convergence

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

Figure 10. Encoding of handles using buffers

equivalence 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 $\xi \in \{\downarrow, \Downarrow\}$.

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 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 $y\,\mathsf{h}\,x$. 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. 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 **handle** generates the translated components of a handle when it is applied to an argument (modulo some BETA(ev)-reductions). Note that, equivalently, we could have defined $h = \lambda z.\mathbf{put}(f',z)$ for the let-binding of h, since $\lambda z.\mathbf{put}(f',z) \sim_\tau \mathbf{thread}\ (\lambda\_.\lambda z.\mathbf{put}(f',z))$ can be proved in $\lambda^\tau(\mathsf{fcb})$, 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 $h\,\mathsf{h}\,\bullet$ we do not know the name f' of the buffer that was previously used to translate $h\,\mathsf{h}\,f$. However, because 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 $h \stackrel{susp}{\Longleftarrow} h$, 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 $h \stackrel{susp}{\Longleftarrow} h \sim h\,\mathsf{h}\,\bullet$ in $\lambda^\tau(\mathsf{fchb})$ and $\lambda^\tau(\mathsf{fch})$, respectively (see Remark 5.9). The $f \stackrel{susp}{\Longleftarrow} \mathbf{get}\ f'$ and the **lazy**-operator in the encoding are necessary, otherwise the future f would force the concurrent evaluation of **get** 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 $T_H : \lambda^\tau(\mathsf{fchb}) \to \lambda^\tau(\mathsf{fcb})$ is compositional, i.e., for all p, D, e, C, we have $T_H(D)[T_H(p)] = T_H(D[p])$ and $T_H(C)[T_H(e)] = T_H(C[e])$.*

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

Proposition 5.2. *An expression $e \in \lambda^\tau(\mathsf{fchb})$ is a $\lambda^\tau(\mathsf{fchb})$-value iff $T_H(e)$ is a $\lambda^\tau(\mathsf{fcb})$-value. A process $p \in \lambda^\tau(\mathsf{fchb})$ is a successful $\lambda^\tau(\mathsf{fchb})$-process iff $T_H(p)$ is a successful $\lambda^\tau(\mathsf{fcb})$-process. For contexts of $\lambda^\tau(\mathsf{fchb})$: $T_H(D)$ ($T_H(E), T_H(F)$, resp.) is a D-context (E-context, F-context, resp.) for $\lambda^\tau(\mathsf{fcb})$ iff D (E, F, resp.) is a D-context (E-context, F-context, resp.) for $\lambda^\tau(\mathsf{fchb})$.*

Because of this proposition, the hard cases for proving convergence equivalence of T_H are the encodings of $\text{HANDLE.NEW}(\mathsf{ev})$- and $\text{HANDLE.BIND}(\mathsf{ev})$-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, hence it is adequate. (In fact, we will prove that ι_H is fully-abstract at the end of this section.) As an immediate consequence of Proposition 4.10 and the adequacy of ι_H we obtain the following correct transformations.

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

Proposition 5.4 ($\downarrow$-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 5.2. 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 5.2) it is easy to see that all reductions of $\lambda^\tau(\mathsf{fchb})$ except for $\text{HANDLE.BIND}(\mathsf{ev})$ and $\text{HANDLE.NEW}(\mathsf{ev})$ 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: $T_H(p) \xrightarrow{\text{FUT.DEREF}(\mathsf{ev})} \xrightarrow{\beta\text{-CBV}(\mathsf{ev})} \xrightarrow{\text{BUFF.PUT}(\mathsf{ev})} \xrightarrow{sp_1} T_H(p')$, where sp_1 is a program transformation in $\lambda^\tau(\mathsf{fcb})$ defined as:

$$\begin{aligned}&(\nu f')(y \overset{\mathit{susp}}{\Longleftarrow} \mathsf{let}\ v = \mathbf{get}\ f'\ \mathsf{in}\ \mathbf{put}(f', v); v\ \mathsf{end}\\ &\quad \mid f'\,\mathsf{b}\,w \mid x \Leftarrow \lambda z.\mathbf{put}(f', z))\\ &\to y \Leftarrow w \mid x \overset{\mathit{susp}}{\Longleftarrow} x\end{aligned}$$

Since these transformations are either ev-reductions or correct (correctness of (sp_1) is proved in (Schwinghammer et al. 2009)), $T_H(p')\downarrow$ 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 $\beta\text{-CBV}(\mathsf{f})$ is the following restriction of $\beta\text{-CBV}(\mathsf{a})$: 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 (Lemma 5.3), we have $T_H(p')\downarrow$ implies $T_H(p)\downarrow$. ☐

Proposition 5.5 ($\downarrow$-reflection of T_H). *For all $\lambda^\tau(\mathsf{fch})$-processes p, the following holds: if $T_H(p)\downarrow$, then $p\downarrow$.*

Proof. Let $p \in \lambda^\tau(\mathsf{fch})$ and $T_H(p)\downarrow$. 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 5.2 implies that p must be successful, too. For the induction step 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 $\text{HANDLE.NEW}(\mathsf{ev})$ or $\text{HANDLE.BIND}(\mathsf{ev})$-reduction. For these cases it holds $p \xrightarrow{\mathsf{ev}} p'$ and $T_H(p') = q$.
- The reduction $T_H(p) \xrightarrow{\mathsf{ev}} q$ is the first reduction of an encoded $\text{HANDLE.BIND}(\mathsf{ev})$ reduction, i.e. it is a $\text{FUT.DEREF}(\mathsf{ev})$-reduction and $p \xrightarrow{\text{HANDLE.BIND}(\mathsf{ev})} p'$. In (Schwinghammer et al. 2009) using Lemma 5.3 we show that there exists a reduction sequence $T_H(p') \xrightarrow{\mathsf{ev},\leq n} q''$, where q'' is successful.
- The reduction $T_H(p) \xrightarrow{\mathsf{ev}} q$ is the first reduction of an encoded $\text{HANDLE.NEW}(\mathsf{ev})$ reduction, i.e. it is a $\beta\text{-CBV}(\mathsf{ev})$-reduction and $p \xrightarrow{\text{HANDLE.NEW}(\mathsf{ev})} p'$. Then again it is possible to construct a reduction sequence $T_H(p') \xrightarrow{\mathsf{ev},\leq n} q''$ where q'' is successful.

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

Proposition 5.6. *The translation T_H is convergence equivalent.*

Proof. Propositions 5.4 and 5.5 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. ☐

Compositionality and convergence equivalence of T_H imply:

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

Since the identity translation $\iota_H : \lambda^\tau(\mathsf{fcb}) \to \lambda^\tau(\mathsf{fchb})$ 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 ι_H are fully abstract.*

Using ι_H and ι_B (see Theorem 4.11) we can define direct translations T'_B, T'_H from $\lambda^\tau(\mathsf{fcb})$ into $\lambda^\tau(\mathsf{fch})$ and vice versa, by setting $T'_B = T_B \circ \iota_H$ and $T'_H = T_H \circ \iota_B$. Since full abstraction and adequacy are preserved under composition, we obtain an adequate translation $T'_B : \lambda^\tau(\mathsf{fcb}) \to \lambda^\tau(\mathsf{fch})$ and a fully abstract translation $T'_H : \lambda^\tau(\mathsf{fch}) \to \lambda^\tau(\mathsf{fcb})$.

Remark 5.9. *Adequacy of T_H (T'_H, resp.) implies that used handles are equivalent to suspended black holes, i.e. $x\,\mathsf{h}\,\bullet \sim x \overset{\mathit{susp}}{\Longleftarrow} x$ 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. $T_H(x\,\mathsf{h}\,\bullet) \equiv x \overset{\mathit{susp}}{\Longleftarrow} x \equiv T_H(x \overset{\mathit{susp}}{\Longleftarrow} x)$.*

$$\begin{aligned}
&\textit{buffer} \triangleq \lambda_.\ \mathsf{let}\ x_g = \mathbf{ref}\ \mathsf{false}, x_p = \mathbf{ref}\ \mathsf{true}, x_s = \mathbf{ref}\ \mathsf{None}\\
&\qquad\qquad \mathsf{in}\ \mathbf{thread}\ \lambda_.\langle x_g, x_p, x_s\rangle\\
&\textit{get} \triangleq \mathbf{thread}(\lambda\, ig.\, \lambda\, x.\\
&\quad \mathsf{let}\ \langle x_g, x_p, x_s\rangle = x,\ getOK = \mathbf{exch}(x_g, \mathsf{false})\\
&\quad \mathsf{in\ if}\ getOK\ \mathsf{then\ let\ Some}\ v = \mathbf{exch}(x_s, \mathsf{None})\\
&\qquad\qquad\qquad\qquad\qquad dummy = \mathbf{exch}(x_p, \mathsf{true}) \quad \mathsf{in}\ v\\
&\qquad \mathsf{else}\ ig\ x)\\
&\textit{put} \triangleq \mathbf{thread}(\lambda\, ip.\, \lambda\, \langle x, v\rangle.\\
&\quad \mathsf{let}\ \langle x_g, x_p, x_s\rangle = x,\ putOK = \mathbf{exch}(x_p, \mathsf{false})\\
&\quad \mathsf{in\ if}\ putOK\ \mathsf{then}\ \mathbf{exch}(x_s, \mathsf{Some}\ v); \mathbf{exch}(x_g, \mathsf{true}); \mathbf{unit}\\
&\qquad \mathsf{else}\ ip\ \langle x, v\rangle)
\end{aligned}$$

Figure 11. Busy-wait encoding of buffers

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 $T_{RB} : \lambda^\tau(\mathsf{fcb}) \to \lambda^\tau(\mathsf{fc})$. Here, the target calculus $\lambda^\tau(\mathsf{fc})$ is like $\lambda^\tau(\mathsf{fcb})$ 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 $\mathsf{option}(\tau)$ with the nullary constructor $\mathsf{None} : \mathsf{option}(\tau)$ and unary constructor $\mathsf{Some} : \tau \to \mathsf{option}(\tau)$. This simplifies the encoding since no dummy values are needed. Fig. 11 shows the encoding of the buffer operations that induces the translation $T_{RB} : \lambda^\tau(\mathsf{fcb}) \to \lambda^\tau(\mathsf{fc})$. On buffers it is defined as follows:

$$\begin{aligned}
T_{RB}(x\ \mathsf{b}\ v) &\triangleq (\nu x_g, x_p, x_s)(x {\Leftarrow} (x_g, x_p, x_s) \mid x_p\ \mathsf{c}\ \mathsf{false}\\
&\qquad\qquad \mid x_g\ \mathsf{c}\ \mathsf{true} \mid x_s\ \mathsf{c}\ \mathsf{Some}\ T_{RB}(v))\\
T_{RB}(x\ \mathsf{b}\ -) &\triangleq (\nu x_g, x_p, x_s)(x {\Leftarrow} (x_g, x_p, x_s) \mid x_p\ \mathsf{c}\ \mathsf{true}\\
&\qquad\qquad \mid x_g\ \mathsf{c}\ \mathsf{false} \mid x_s\ \mathsf{c}\ \mathsf{None})
\end{aligned}$$

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 $id : \lambda^\tau(\mathsf{fc}) \to \lambda^\tau(\mathsf{fch})$ show that the reduction rules of $\lambda^\tau(\mathsf{fch})$, except for CELL.EXCH(ev) and the handle-rules, and the transformations of Fig. 7, are also correct transformations for $\lambda^\tau(\mathsf{fc})$. This allows us to prove convergence equivalence of $T_{RB}(p)$, which implies correctness:

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

As a corollary, using the translation $T'_H : \lambda^\tau(\mathsf{fch}) \to \lambda^\tau(\mathsf{fcb})$ described at the end of the previous section, we obtain an observationally correct and adequate translation $T_{RB} \circ T'_H$ from $\lambda^\tau(\mathsf{fch})$ into $\lambda^\tau(\mathsf{fc})$.

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'_H, T'_B 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 $\lambda^\tau(\mathsf{fc})$ to the calculus $\lambda(\mathsf{fh})$. We achieve this in two steps: First, the data constructors and case-expressions of $\lambda^\tau(\mathsf{fc})$ 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 $\lambda(\mathsf{fh})$ (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 equivalent 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_H, T_B that encode blocking by waiting and queuing and vice versa, improve upon the busy-wait encoding? And, is the translation T_B 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 Birkedal 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

A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. *POPL'09*, pages 340–353. ACM Press, 2009.

N. Bohr and L. Birkedal. Relational reasoning for recursive types and references. *APLAS'06*, LNCS 4279, pages 79–96. Springer, 2006.

A. Carayol, D. Hirschkoff, and D. Sangiorgi. On the representation of McCarthy's amb in the pi-calculus. *TCS*, 330(3):439–473, 2005.

R. De Nicola and M. Hennessy. Testing equivalences for processes. *TCS*, 34:83–133, 1984.

C. Fournet, F. Le Fessant, L. Maranget, and A. Schmitt. JoCaml: A language for concurrent distributed and mobile programming. In *Advanced Functional Programming*, LNCS 2638, pages 129–158. Springer, 2002.

D. Gorla. Towards a unified approach to encodability and separation results for process calculi. *CONCUR '08*, pages 492–507, Springer, 2008.

A. Jeffrey and J. Rathke. A theory of bisimulation for a fragment of concurrent ML with local names. *TCS*, 323(1-3):1–48, 2004.

V. Koutavas and M. Wand. Small bisimulations for reasoning about higher-order imperative programs. *POPL'06*, pages 141–152. ACM, 2006.

X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *J. Autom. Reasoning*, 41 (1):1–31, 2008.

B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. *PLDI'88*, pages 260–267. ACM Press, 1988.

J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda calculus with futures. *TCS* 364(3):338–356, 2006.

J. Niehren, D. Sabel, M. Schmidt-Schauß, and J. Schwinghammer. Observational semantics for a concurrent lambda calculus with reference cells and futures. *23rd MFPS*, ENTCS 173, pages 313–337. Elsevier, 2007.

C.-H. L. Ong. Non-determinism in a functional setting. In *LICS '93*, pages 275–286. IEEE Computer Society, 1993.

J. Parrow. Expressiveness of process algebras. *ENTCS*, 209:173–186, 2008.

S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. *POPL'96*, pages 295–308. ACM Press, 1996.

U. S. Reddy and H. Yang. Correctness of data representations involving heap data structures. *Sci. Comput. Program.*, 50(1–3):129–160, 2004.

A. Rensink and W. Vogler. Fair testing. *Inform. and Comput.*, 205(2):125–198, 2007.

J. H. Reppy. *Concurrent Programming in ML.* CUP, 1999.

J. C. Reynolds. Types, abstraction, and parametric polymorphism. In *Information Processing '83*, pages 513–523. North-Holland, 1983.

J. G. Riecke. Fully abstract translations between functional languages. *POPL'91*, pages 245–254, ACM Press, 1991.

E. Ritter and A. M. Pitts. A fully abstract translation between a lambda-calculus with reference types and standard ml. *TLCA'95*, LNCS 902, Springer, 1995.

A. Rossberg, D. Le Botlan, G. Tack, T. Brunklaus, and G. Smolka. *Trends in Functional Programming*, volume 5, chapter Alice Through the Looking Glass, pages 79–96. Munich, Germany, 2006.

D. Sabel and M. Schmidt-Schauß. A call-by-need lambda calculus with locally bottom-avoiding choice: context lemma and correctness of transformations. *Math. Struct. Comp. Sci.*, 18(3):501–553, 2008.

M. Schmidt-Schauß, J. Niehren, D. Sabel, and J. Schwinghammer. Adequacy of compositional translations for observational semantics. In *5th IFIP*, IFIP 273, pages 521–535. Springer, 2008.

M. Schmidt-Schauß and D. Sabel. On generic context lemmas for lambda calculi with sharing. Frank report 27, J.W.Goethe-University, Frankfurt, 2007.

M. Schmidt-Schauß, J. Niehren, J. Schwinghammer, and D. Sabel. Adequacy of compositional translations for observational semantics. Frank report 33, J.W.Goethe-University, Frankfurt, 2009.

J. Schwinghammer, D. Sabel, J. Niehren, and M. Schmidt-Schauß. On correctness of buffer implementations in a concurrent lambda calculus with futures. Frank report 37, J.W.Goethe-University, Frankfurt, 2009.