Program Equivalence for a Concurrent Lambda Calculus with Futures

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

[1] INRIA Futurs, Lille, France, Mostrare Project
[2] J. W. Goethe-Universität, Frankfurt, Germany
[3] Saarland University, Programming Systems Lab, Saarbrücken, Germany

Technical Report Frank-26

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

October 14, 2006

Abstract. Reasoning about the correctness of program transformations requires a notion of program equivalence. We present an observational semantics for the concurrent lambda calculus with futures $\lambda(\mathsf{fut})$, which formalizes the operational semantics of the programming language Alice ML. We show that natural program optimizations, as well as partial evaluation with respect to deterministic rules, are correct for $\lambda(\mathsf{fut})$. This relies on a number of fundamental properties that we establish for our observational semantics.

1 Introduction

Alice ML [16] is a concurrent functional programming language of the ML family [3, 10] that is inspired by the concurrent constraint programming language Mozart-Oz [20]. In this paper we present the first observational semantics for Alice ML, which is needed to justify correctness of program transformations.

Alice ML features concurrent programming with mixed eager and lazy threads, which may be distributed in a network transparent fashion. Alice ML follows the tradition of strong static typing, and extends it with dynamic components supporting typed open programming [15]. All synchronization is based on futures [1], a restricted form of logic variables. Alice ML can express channels as in the asynchronous π-calculus [9], streams as in dataflow languages, and JoCaml-like joins [4].

The operational semantics of the Alice ML core language is defined in [12] by the concurrent lambda calculus with futures $\lambda(\mathsf{fut})$. This calculus extends on

previous lambda calculi with futures [6] that are only intended to model parallel execution of purely functional programs, by adding a process level on top of the call-by-value lambda calculus. Each process consists of a set of concurrent threads $x \Leftarrow e$ that may eventually assign the value of lambda term e to future x. The future x itself may occur in the expression e so that recursive bindings on top level can easily be expressed. Operations that "need" the value of an undetermined future block until its value becomes available. In contrast, applications with futures as arguments succeed and may proceed irrespective of the future's status. This leads to a convenient form of (automatic) data-driven synchronization.

Reference cells with atomic value exchange are the only *observably* nondeterministic construct of $\lambda(\mathsf{fut})$. In their presence it becomes more difficult to argue the correctness of program transformations, as needed for justifying partial evaluation or future optimisation by touch elimination [6]. Notions of program equivalence for deterministic languages with reference cells based on contextual equivalence, as e.g. considered in [13], have to be extended to take non-determinism and concurrency into account. Most previous work in this area focuses on process calculi [9, 19] rather than high-level languages, or investigates the theory obtained by encoding lambda calculi into process calculus (for instance, [18]). In [5, 7], program behavior in fragments of Concurrent ML is characterized by bisimilarity with respect to a labelled transition system. However, program equivalences for languages with future-based concurrency, rather than channels, have not been considered in the literature.

In this paper we present an observational semantics for $\lambda(\mathsf{fut})$, which defines two processes to be equivalent if they exhibit the same termination behavior in all contexts. A successfully terminated process, informally, has no pending evaluations, and every future x references a non-variable value or a process variable, perhaps through several indirections. Since the calculus is non-deterministic we combine may- and must-convergence, a modelling technique also used for lambda-calculi with *amb* [2, 11, 17]. A process is may-convergent if it has a reduction to a successfully terminated process; it is must-convergent if every reduction-descendant is may-convergent. Examples of processes that cannot successfully terminate include cyclic chains of future-references, deadlock-like situations, and strongly divergent processes (i.e., those that are not may-convergent).

This definition of observational equivalence treats all "erroneous" processes as equivalent, and distinguishes processes that may lead to an error from those that do not. A similar combination of may- and must-convergence is also known from the use of convex powerdomains in domain-theoretic models [14]. However, these models are usually far from being fully abstract, and powerdomains by themselves do not provide for a treatment of concurrency and synchronization, dynamically created threads and state, or sharing.

By only taking strong divergence into account it follows that every (possibly non-terminating) $\lambda(\mathsf{fut})$ process that keeps the chance to successfully terminate indefinitely is not erroneous. Consequently, whenever a process diverges this is already witnessed by fair reduction, and hence fair program equivalence agrees with the unconstrained equivalence (see also [2, 17]).

$$
\begin{array}{l}
x, y, z \in \mathit{Var} \\
\quad c \in \mathit{Const} ::= \mathbf{unit} \mid \mathbf{cell} \mid \mathbf{thread} \mid \mathbf{handle} \mid \mathbf{lazy} \\
\quad\quad e \in \mathit{Exp} ::= x \mid c \mid \lambda x.e \mid e_1\ e_2 \mid \mathbf{exch}(e_1, e_2) \\
\quad\quad v \in \mathit{Val} ::= x \mid c \mid \lambda x.e \\
p \in \mathit{Process} ::= p_1 \mathsf{I}\ p_2 \mid (\nu x)p \mid x\,\mathsf{c}\,v \mid x \Leftarrow e \mid y\,\mathsf{h}\,x \mid y\,\mathsf{h}\,\bullet \mid x \overset{susp}{\Longleftarrow} e
\end{array}
$$

Fig. 1. Expressions and processes of $\lambda(\mathsf{fut})$

After introducing the lambda calculus $\lambda(\mathsf{fut})$ and its operational semantics in the next section, Section 3 defines an observational equivalence on $\lambda(\mathsf{fut})$ programs that reflects the non-deterministic reduction. We prove a context lemma which shows that observations in evaluation contexts suffice to characterize program equivalence. Section 4 establishes the correctness of partial evaluation and several program transformations. In particular, we show that $\lambda(\mathsf{fut})$ is internally consistent, in the sense that all deterministic reduction rules can be used as program transformations. It is also shown that the only potentially non-deterministic reduction rule – value exchange in cells – is correct if there is no ambiguous execution possibility. This is the maximal amount of partial evaluation that can be expected for a non-deterministic calculus. Further examples of transformations that we prove correct include a "garbage collection" rule, "path compression", and "deep" beta-reduction. The main tool in proving equivalences, besides the context lemma, are *complete sets of forking* and *commuting diagrams*, adapted from [8, 17].

2 Lambda(Fut)

We recall the $\lambda(\mathsf{fut})$ calculus from [12] including lazy futures. We deviate from *loc. cit.* by considering an untyped variant, and substituting a sharing beta rule that lends itself to easier proofs than the usual one. Nevertheless, the full beta rule will be shown correct in Section 4.

Syntax. The syntax of $\lambda(\mathsf{fut})$ has two levels, the layer of λ-calculus expressions $e \in \mathit{Exp}$ for sequential functional computation inside of threads, and the layer of processes $p \in \mathit{Process}$ for concurrent computation composing multiple sequential threads in parallel. It is defined in Fig. 1.

The expressions of $\lambda(\mathsf{fut})$ are standard λ-terms, new operations are introduced by (higher-order) constants: **unit** is a dummy value, and constants **thread**, **lazy** and **handle** serve for introducing futures simultaneously with threads, suspended computations, or handles that can bind their values, respectively. The constant **cell** serves for introducing reference cells, and $\mathbf{exch}(e_1, e_2)$ for an atomic exchange of cell values. Values v are defined as usual in a call-by-value λ-calculus, and consist of variables, constants and abstractions. Examples may use syntax sugar, like $\lambda\_.e$ instead of $\lambda x.e$ where x does not appear in e.

Processes p are reminiscent of π-calculus terms. They are built from basic components by parallel composition $p_1 \mathsf{I}\ p_2$ and new name operators $(\nu x)p$. A

$$p_1 \mid p_2 \;\equiv\; p_2 \mid p_1 \qquad (p_1 \mid p_2) \mid p_3 \;\equiv\; p_1 \mid (p_2 \mid p_3)$$
$$(\nu x)(\nu y)p \equiv (\nu y)(\nu x)p \qquad (\nu x)(p_1) \mid p_2 \equiv (\nu x)(p_1 \mid p_2) \quad \text{if } x \text{ does not occur in } p_2$$

Fig. 2. Structural congruence of processes

structural congruence $\equiv$ on processes is defined as the least congruence relation containing the identities in Fig. 2. The first two axioms render parallel composition associative and commutative. The final *scope extrusion* rule is used to extend the scope of a local variable.

A *component* p is a process without parallel composition or name restrictions. A *component of a process* p is a component p_1 such that $p \equiv (\nu \overline{x})(p_1 \mid p_2)$ for some variables $\overline{x}$ and process p_2. We distinguish five types of components: a thread $x \Leftarrow e$ is a concurrent component whose evaluation will eventually bind the future x to the value of expression e unless it diverges or suspends. We call such variables x *concurrent futures*. Analogously, a lazy computation $x \overset{susp}{\Longleftarrow} e$ is a component that will eventually bind the (lazy) future x to the value of e, but *only if evaluation is triggered*, by some active thread suspending on x. Such variables x are *lazy futures*. A cell $x \,\mathsf{c}\, v$ associates a (memory location) name x to a value v. A handle component $y \,\mathsf{h}\, x$ associates a handle y to a future 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 $y \,\mathsf{h}\, \bullet$ means that y is a handle that has already been used to bind its future. A process p *introduces a variable* x if p contains some component of the following form for some y, e, v.

$x \Leftarrow e$	x is a *concurrent future* (for e)
$x \overset{susp}{\Longleftarrow} e$	x is a *lazy future* (for e)
$x \,\mathsf{c}\, v$	x is a *cell* (with content v)
$x \,\mathsf{h}\, y$	x is an *unused handle* (for future y)
$y \,\mathsf{h}\, x$	x is a *handled future* (handled by y)
$x \,\mathsf{h}\, \bullet$	x is a *used handle*

Introduced variables are also called *process variables*. A process is *well-formed* iff it does not introduce any variable twice, and cells $x \,\mathsf{c}\, v$ contain only values.

Free and bound variables of expressions and processes are defined as usual; the only scope bearing constructs are λ-binder and new operators (νx). We identify expressions and processes up to consistent renaming of bound variables. We write $\mathsf{fv}(p)$ and $\mathsf{fv}(e)$ for the free variables of a process and expression, respectively, and $e[e'/x]$ for the (capture-free) substitution of e' for x in e. Whenever it is appropriate and can be done without ambiguities, we will use the distinct variable convention, i.e., we assume that all bound variables in expressions and processes are distinct, and that free variables are distinct from bound variables. However, at several places in this paper, we have to explicitly take care of the names of bound variables.

The binding operator ν can be seen as defining the observational scope of variables. Using the distinct variable convention and moving ν-binders to the

top-level, a process p that introduces variables $\overline{x}\,\overline{y}$ can be written in the form $(\nu\overline{x})p$, where p does not contain further ν-binders. The (free) variables $y \in \overline{y}$ are interpreted as being directly observable by an external observer. Variables with a restricted scope may still be observable indirectly.

Operational Semantics. A *context* is a process with exactly one occurrence of the hole marker, i.e. the special constant $[\,]$. The hole marker cannot occur at the positions that are reserved for variable introduction, and in a cell $x\,\mathsf{c}\,v$, the position of the hole can only be in e for $v = \lambda x.e$. Let γ be a context, and η be a term or a process that can be plugged into its hole, then we write $\gamma[\eta]$ for the result of replacing $[\,]$ in γ by η (possibly capturing free variables of η).

A context C is a context of type expression, i.e., a process where the hole marker $[\,]$ occurs in expression position. We call C *flat* if its hole does not occur below a lambda binder, and *deep* otherwise. A context D is a context of type process, where the hole marker occurs in process position. In Fig. 3 we define particular flat contexts of type expression that we call *evaluation contexts* (ECs) E and *future ECs* F. ECs encode the standard call-by-value, left-to-right reduction strategy, while future ECs control dereferencing operations on futures: dereferencing is only allowed when the value of the future is needed.

We define the operational semantics of $\lambda(\mathsf{fut})$ using a (small-step) reduction denoted by $\rightarrow$, or $\xrightarrow{\mathsf{ev}}$ in case we want to distinguish it from the general transformations introduced in Section 4. It is the least binary relation on processes $p \rightarrow p'$ satisfying the rules in Fig. 4.

By rule (THREAD.NEW(ev)), evaluation of $\mathbf{thread}\,\lambda x.e$ spawns a new thread $x \Leftarrow e$, where x may occur in e (so it may be viewed as a recursive equation $x = e$, but directed from right to left). Similarly, $\mathbf{lazy}\,\lambda x.e$ creates a new suspended computation $x \overset{susp}{\Longleftarrow} e$. Both dereferencing of fully evaluated futures and triggering of suspended computations is controlled by the use of contexts F in (FUT.DEREF(ev)) and (LAZY.TRIGGER(ev)), resp. The lazy variant (LBETA(ev)) of beta reduction also takes advantage of this machinery. In (HANDLE.NEW(ev)) the expression $\mathbf{handle}\,\lambda x.\lambda y.e$ introduces a handle component $y\,\mathsf{h}\,x$ with static scope in e; the application $y\,v$ in (HANDLE.BIND(ev)) "consumes" the handle y and binds x to v, resulting in a used handle $y\,\mathsf{h}\,\bullet$ and thread $x \Leftarrow v$. Reducing $\mathbf{cell}\,v$ with rule (CELL.NEW(ev)) creates a new cell $z\,\mathsf{c}\,v$ with contents v. The exchange operation $\mathbf{exch}(z, v')$ writes v' to the cell and returns the previous contents. Since this is an *atomic* operation, no other thread can interfere.

Note that reduction for a given process may be non-deterministic. Note also that reduction preserves well-formedness as well as non-well-formedness. For a reduction or transformation t, t^+, t^*, t^ϵ denotes the transitive, reflexive-transitive and reflexive closure of t, respectively.

Example 2.1. In $\lambda(\mathsf{fut})$ we can define a binary non-deterministic choice operator that receives two values as arguments, and non-deterministically selects one of them. Let $K_1 = \lambda x \lambda y.x$ and $K_2 = \lambda x \lambda y.y$, and let *choice* be

$$\lambda s \lambda t.\mathit{let}\ z = \mathbf{cell}\ K_1\ \mathit{in}\ \mathit{let}\ y = \mathbf{thread}\ (\lambda\_.\mathbf{exch}(z, K_2))\ \mathit{in}\ (\mathbf{exch}(z, K_2))\,s\,t$$

ECs	$E ::= x{\Leftarrow}\tilde{E}\ ,$	$\tilde{E} ::= [\,] \mid \tilde{E}\,e \mid v\,\tilde{E} \mid \mathbf{exch}(\tilde{E}, e) \mid \mathbf{exch}(v, \tilde{E})$
Future ECs	$F ::= x{\Leftarrow}\tilde{F}\ ,$	$\tilde{F} ::= \tilde{E}[[\,]\,v] \mid \tilde{E}[\mathbf{exch}([\,], v)]$
Process ECs	$D ::= [\,] \mid p \mid D \mid D \mid p \mid (\nu x)D$	

Fig. 3. Evaluation contexts

Reduction rules.

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

Distinct variable convention. We assume that all processes to which a rules apply satisfy the distinct variable convention, and that all new binders use fresh variables (z and z'). Reduction results will then satisfy the distinct variable convention as well, except for the rule FUT.DEREF(ev) where values with bound variables can get copied. In this case, α-renaming has to be performed before applying the next rule.

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

Fig. 4. One-step reduction relation of $\lambda(\mathsf{fut})$ denoted by $\to$ or $\overset{\mathsf{ev}}{\to}$

where *let* is just syntax sugar. One can verify that $E[\mathit{choice}\,v_1\,v_2] \mid p$ may reduce to $E[v_i] \mid p \mid \ldots$, for any p and $i = 1, 2$, which is equivalent to $E[v_i] \mid p$, using correctness of GC(see Theorem 4.11).

3 Program Equivalence

Two processes are equivalent if it is impossible for an observer to distinguish them. We model the observer by contexts, which test whether or not a process in that context may or must terminate successfully.

A process p is *successful*, i.e. has terminated successfully, iff for every component $x{\Leftarrow}e$ of p, the future x is bound (possibly via a chain $x{\Leftarrow}x_1 \mid x_1{\Leftarrow}x_2 \mid \ldots \mid x_{n-1}{\Leftarrow}x_n$) to a non-variable value, a cell, a lazy future, a handle, or a handled future. For example, $x{\Leftarrow}\lambda y.y$ is successful, while $x{\Leftarrow}x$ (a "black hole") and $x{\Leftarrow}yx \mid y{\Leftarrow}xy$ (a deadlocked process) are not successful.

Let p be a process. We say that p is *may-convergent* ($p{\downarrow}$) if there exists a sequence of reductions $p \to^* p'$ such that p' is successful. It is *must-convergent* ($p{\Downarrow}$) if all reduction descendants p' of p are may-convergent. This implies that all irreducible descendants of p must succeed. We say that p is *must-divergent* ($p{\Uparrow}$) if it has no reduction descendant that succeeds. It is *may-divergent* ($p{\uparrow}$) if some reduction descendant of p is must-divergent. With $\mathrm{Suc}(p)$ ($\mathrm{Div}(p)$, resp.) we denote all sequences of reductions for p that end in a successful process (must-divergent process, resp.). Note that all processes p satisfy $p{\Uparrow} \Leftrightarrow \neg p{\downarrow}$ and $p{\uparrow} \Leftrightarrow \neg p{\Downarrow}$, and that non-well-formed processes are must-divergent.

Lemma 3.1. *If p contains a future cycle $x_1{\Leftarrow}x_2 \mid x_2{\Leftarrow}x_3 \mid \ldots \mid x_n{\Leftarrow}x_1$ then $p{\Uparrow}$. In particular, if $p{\downarrow}$ then p does not contain such a cycle.*

We write $\downarrow$ for the set of may-convergent processes, and $\Downarrow$ for the set of must-convergent processes. Let $P = \downarrow$ or $P = \Downarrow$. We define binary relations $\leq_P$ both for processes and expressions, such that for all $p, p' \in \mathit{Process}$ and $e, e \in \mathit{Exp}$:

$$\begin{array}{lll} p \leq_P p' & \text{iff} & \forall D.\ D[p] \in P \Rightarrow D[p'] \in P \\ e \leq_P e' & \text{iff} & \forall C.\ C[e] \in P \Rightarrow C[e'] \in P \end{array}$$

Note that $e \leq_P e'$ iff $C[e] \leq_P C[e']$ for all C. The *contextual preorder* $\leq$ is the intersection of may- and must-contextual approximation $\leq_\downarrow$ and $\leq_\Downarrow$. *Contextual equivalence* $\sim$ is the equivalence relation $\leq \cap \geq$ induced by the contextual preorder $\leq$. It is easy to see that contextual equivalence $\sim$ on expressions is a congruence, i.e., $\sim$ is an equivalence relation such that $e \sim e'$ implies $C[e] \sim C[e']$ for all contexts C. Note that neither may- nor must-convergence alone yields a satisfactory notion of observational equivalence, as the former cannot distinguish v from $\mathit{choice}\, v\, \bot$ while the latter equates $\mathit{choice}\, v\, \bot$ and $\bot$, where v is any value and $\bot$ e.g. represents non-convergence, e.g. a future x with thread $x{\Leftarrow}x$.

In Appendix B we show a context lemma for processes claiming that for equivalence proofs we can assume that ν-binders are shifted upwards. Here we present the more fundamental context lemma for expressions, stating that ECs provide already enough observations to distinguish observationally nonequivalent terms. Establishing equivalences is made considerably more tractable by the context lemma. We define contextual approximations in evaluation contexts for the set of processes $P = \downarrow$ and $P = \Downarrow$.

$$e \leq_P^{\mathsf{ev}} e' \ \text{ iff } \ \forall E \forall D : D[E[e]] \in P \Rightarrow D[E[e']] \in P$$

Proposition 3.2 (Context Lemma). *For all $e_1, e_2 \in \mathit{Exp}$:*

$$e_1 \leq_\downarrow^{\mathsf{ev}} e_2 \ \ \textit{and} \ \ e_1 \leq_\Downarrow^{\mathsf{ev}} e_2 \quad \Rightarrow \quad e_1 \leq e_2$$

We only give an outline of the proof here. In a first step, we prove a context lemma for may-convergence. This requires to generalize to contexts with several holes. The proof is by induction on the lexicographic measure that counts the number of steps until successful termination in the first component and the

number of holes in the second. In a second step the case of must-convergence is shown by using the validity of the context lemma for may-convergence.

The full proof can be found in Appendix C. In Appendix D we show that may- and must-convergence do not change if reductions are restricted to be fair.

4 Program Transformations

We present a set of transformation rules that allow for partial evaluation, and show which of these reduction rules are correct. Most importantly, we show that call-by-value beta reduction can be performed in arbitrary contexts.

Candidates of transformation rules are collected in Fig. 5. They are parametrized by strategies strat which fix the contexts in which the rule can be applied. We assume all transformations to be closed under structural congruence and process ECs. The strategy ev is the reduction strategy of $\lambda(\mathsf{fut})$. It permits ECs E for all rules but FUT.DEREF and LAZY.TRIGGER where it requires future ECs F. The strategy f permits all flat contexts, while d insists on deep contexts. Other strategies can be defined by Boolean combinations, for instance $\neg\mathsf{ev}\wedge\mathsf{f}$. In particular, the strategy with arbitrary contexts is $\mathsf{a} = \mathsf{f} \vee \mathsf{d}$.

The first set of transformation rules in Fig. 5 is obtained by lifting reduction rules of $\lambda(\mathsf{fut})$ from ECs to contexts permitted by the strategy. The second set is most important. It contains call-by-value β-reduction in contexts permitted by the strategy, garbage collection, and deterministic cell exchange. The dereferencing of values into cells (CELL.DEREF) is included mainly for technical reasons.

Definition 4.1. *A transformation t is* correct *iff $(p, p') \in t$ implies that $p \sim p'$.*

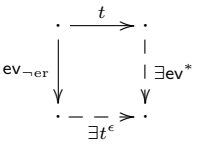
Proposition 4.2. *The following transformations are* not *correct:* CELL.EXCH(ev), THREAD.NEW($\neg\mathsf{ev}$) HANDLE.NEW($\neg\mathsf{ev}$), HANDLE.BIND($\neg\mathsf{ev}$), CELL.NEW($\neg\mathsf{ev}$), LAZY.NEW($\neg\mathsf{ev}$), *and* LAZY.TRIGGER(f). *Also a rule* β-CBN *with* $C[(\lambda x.e)\ e'] \ \to\ C[e[e'/x]]$ *would be incorrect.*

Lemma 4.3. *A transformation t that is closed under all contexts D is correct iff it satisfies $p{\downarrow} \Leftrightarrow p'{\downarrow}$ and $p{\uparrow} \Leftrightarrow p'{\uparrow}$ for all pairs $(p, p') \in t$.*

4.1 Correctness of Deterministic Reductions

We show that all deterministic reduction steps of $\lambda(\mathsf{fut})$ are correct transformations. This excludes the rule CELL.EXCH(ev), the only source of non-determinism in $\lambda(\mathsf{fut})$. The proof relies on the diagrams used in [12] to show the uniform confluence of the fragment of $\lambda(\mathsf{fut})$ without cell exchange and handle errors.

Let ev be the reduction $\xrightarrow{\mathsf{ev}}$ of $\lambda(\mathsf{fut})$ and $p \xrightarrow{\mathsf{ev}_{\neg\mathrm{er}}} p'$ iff $p \xrightarrow{\mathsf{ev}} p'$ and p is may-convergent.



Lemma 4.4. *A transformation t on processes that is closed under D-contexts and subsumed by reduction $t \subseteq \mathsf{ev}$ is correct if it satisfies the forking condition* $\mathsf{ev}_{\neg er}^{-1} \circ t \subseteq\ t^{\epsilon} \circ (\mathsf{ev}^*)^{-1}$.

Lifting reduction to transformation rules.

$$
\begin{array}{ll}
(\text{LBETA}(\mathsf{strat})) & C[(\lambda y.e)\, v] \to (\nu y)(C[e] \mid y \Leftarrow v) \\
(\text{THREAD.NEW}(\mathsf{strat})) & C[\mathbf{thread}\, v] \to (\nu z)(C[z] \mid z \Leftarrow v\, z) \\
(\text{FUT.DEREF}(\mathsf{strat})) & C[x] \mid x \Leftarrow v \to C[v] \mid x \Leftarrow v \\
(\text{HANDLE.NEW}(\mathsf{strat})) & C[\mathbf{handle}\, v] \to (\nu z)(\nu z')(C[v\, z\, z'] \mid z'\, \mathsf{h}\, z) \\
(\text{HANDLE.BIND}(\mathsf{strat})) & C[x\, v] \mid y\, \mathsf{h}\, x \to C[\mathbf{unit}] \mid y \Leftarrow v \mid x\, \mathsf{h}\, \bullet \\
(\text{CELL.NEW}(\mathsf{strat})) & C[\mathbf{cell}\, v] \to (\nu z)(C[z] \mid z\, \mathsf{c}\, v) \\
(\text{CELL.EXCH}(\mathsf{strat})) & C[\mathbf{exch}(y, v_1)] \mid y\, \mathsf{c}\, v_2 \to C[v_2] \mid y\, \mathsf{c}\, v_1 \\
(\text{LAZY.NEW}(\mathsf{strat})) & C[\mathbf{lazy}\, v] \to (\nu z)(C[z] \mid z \overset{susp}{\Leftarrow} v\, z) \\
(\text{LAZY.TRIGGER}(\mathsf{strat})) & C[x] \mid x \overset{susp}{\Leftarrow} e \to C[x] \mid x \Leftarrow e
\end{array}
$$

Call-by-value beta reduction and other deterministic transformations.

$$
\begin{array}{ll}
(\beta\text{-CBV}(\text{strat})) & C[(\lambda x.e)\ v] \ \to\ C[e[v/x]] \\
(\text{CELL.DEREF}) & p \mid y\, \mathsf{c}\, x \mid x \Leftarrow v \to\ p \mid y\, \mathsf{c}\, v \mid x \Leftarrow v \\
(\text{GC}) & p \mid (\nu y_1) \ldots (\nu y_n) p' \ \to\ p \\
 & \text{if } p' \text{ is successful and } y_1, \ldots, y_n \text{ contains all process variables of } p' \\
(\text{DET.EXCH}) & (\nu x)(y \Leftarrow \tilde{E}[\mathbf{exch}(x, v_1)] \mid x\, \mathsf{c}\, v_2) \to\ (\nu x)(y \Leftarrow \tilde{E}[v_2] \mid x\, \mathsf{c}\, v_1)
\end{array}
$$

No capturing. The same conditions as in Fig. 4 are assumed. In addition we assume that no variables are moved out of their scope or into the scope of some other binder, i.e., $\mathsf{fv}(v) \cap \mathsf{bv}(C) = \emptyset$, and that α-renaming is also done after CELL.DEREF.

Closure. Transformations are always closed under structural congruence and D contexts. For all above rules $r(\mathsf{strat})$ we write $p_1 \xrightarrow{r(\mathsf{strat})} p_2$ if $p_1 \to p_2$ by this rule.

Fig. 5. Transformation rules for some strategy strat permitting contexts C

Proof. Since t is closed under D-contexts it suffices to show for all $(p, p') \in t$ that p and p' have identical may- and must-convergence behaviour. That $p'{\downarrow} \Rightarrow p{\downarrow}$ and $p'{\uparrow} \Rightarrow p{\uparrow}$ is obvious since $t \subseteq \mathsf{ev}$. We prove the remaining cases:

$\mathbf{p}{\downarrow} \Rightarrow \mathbf{p'}{\downarrow}$: By induction on the length of $R \in \mathrm{Suc}(p)$. This length cannot be 0 since $t \subseteq \mathsf{ev}$. We prove the induction step. Note that p is may-convergent since the reduction sequence exists. Thus, we can apply the forking diagram to the first reduction step. If the diagram is closed by an $\xrightarrow{=}$ step, then $p'{\downarrow}$. Otherwise the induction hypothesis applies to the result of the first reduction step.

$\mathbf{p}{\uparrow} \Rightarrow \mathbf{p'}{\uparrow}$: By induction on the length of a minimal reduction sequence $R \in \mathrm{Div}(P)$. If the length is 0, then p is must-divergent and so $p'{\Uparrow}$, since we have already established $p'{\downarrow} \Rightarrow p{\downarrow}$. Otherwise, p is may-convergent, so that we can apply the forking diagram. The rest follows from the induction hypothesis. □

Note that t preserves must-divergence since reduction ev does. If t raises a handle error, i.e. generates components of the form $E[z\ v_1] \mid z\, \mathsf{h}\, \bullet$, then the result is a must-divergent process.

Proposition 4.5. *All reduction steps of* $\lambda(\mathsf{fut})$ *are correct program transformations except for* CELL.EXCH(ev).

Proof. The diagrams required by Lemma 4.4 can be shown as in [12], with a slight modification: Instead of call-by-value beta reduction one needs to consider LBETA(ev) and additionally the overlappings with rule CELL.NEW(ev). Both modifications are easy to handle. The only rule for which some care is needed is the rule HANDLE.BIND(ev). This rule can introduce non-determinism, but only when raising handle errors which results in a must-divergent process: a typical counter example is $E_1[z\ v_1] \mid E_2[z\ v_2] \mid z\,\mathsf{h}\,y$ which has two reducts $E_1[\mathbf{unit}] \mid y{\Leftarrow}v_1 \mid E_2[z\ v_2] \mid z\,\mathsf{h}\bullet$ and $E_1[z\ v_1] \mid y{\Leftarrow}v_2 \mid E_2[\mathbf{unit}] \mid z\,\mathsf{h}\bullet$ that cannot be joined, but both constitute handle-errors. The rule commutes with itself in case no handle error is raised. □

4.2 Correctness of LBETA(f), FUT.DEREF(f), GC and DET.EXCH

Lemma 4.6. *A transformation t on processes is correct if it satisfies the following three conditions:*

(**fork**) $\mathsf{ev}^{-1} \circ t \ \subseteq\ t^* \circ (\mathsf{ev}^\epsilon)^{-1}$ (**commute**) $t \circ \mathsf{ev} \ \subseteq\ \mathsf{ev}^* \circ t^\epsilon$
(**success**) *for all $(p,p') \in t$: p is successful iff p' is successful and $(p,p') \notin \mathsf{ev}$.*

Proof. Since t is closed under D-contexts, it is sufficient to show for all p, p' with $(p,p') \in t$ that p and p' have identical may- and must-convergence behaviour.

$\mathbf{p}{\downarrow} \Rightarrow \mathbf{p}'{\downarrow}$: By induction on the length of $R \in \mathrm{Suc}(p)$, we show that there exists $R' \in \mathrm{Suc}(p')$ of smaller or equal length. In the base case of length 0, p is successful and thus p' by condition (**success**). Otherwise consider the first reduction step. There exists p_1 such that $p \xrightarrow{\mathsf{ev}} p_1$ and p_1 has a smaller successful reduction sequence. Thus, we can apply condition (**fork**) for some p_1' with $p' \xrightarrow{\mathsf{ev}^\epsilon} p_1'$ and $p_1 \xrightarrow{t^*} p_1'$. We conclude the proof by induction on the length of $p_1 \xrightarrow{t^*} p_1'$. If this length is 0 then $p_1 \equiv p_1'$ so p' has a successful reduction sequence of length smaller or equal to that of p. If the length is n, we apply the first induction hypothesis to the first transformation step, and use the other induction hypothesis for the remaining sequence of $n-1$ transformation steps.

$\mathbf{p}'{\downarrow} \Rightarrow \mathbf{p}{\downarrow}$: By induction on the length of $R \in \mathrm{Suc}(p')$. The case $p \xrightarrow{\mathsf{ev}} p'$ is obvious, so we can assume $(p,p') \notin \mathsf{ev}$. In the base case, this length is 0 so p' is successful. Assumption (**success**) implies that p is successful too. For larger lengths, we can apply the (**commute**) condition, and then the induction hypothesis.

$\mathbf{p}{\uparrow} \Rightarrow \mathbf{p}'{\uparrow}$: By induction on the length of $R \in \mathrm{Div}(p)$ we show that there exists $R' \in \mathrm{Div}(p')$ of smaller or equal length. In the base case, $p{\Uparrow}$, hence $p'{\Uparrow}$ as shown in case $\mathbf{p}'{\downarrow} \Rightarrow \mathbf{p}{\downarrow}$. The induction step uses the (**fork**) diagram.

$\mathbf{p}'{\uparrow} \Rightarrow \mathbf{p}{\uparrow}$: By induction on the length of $R \in \mathrm{Div}(p')$. In the base case, p' must-diverges and so does p as we showed in case $\mathbf{p}{\downarrow} \Rightarrow \mathbf{p}'{\downarrow}$. The induction step relies on the (**commute**) diagram. □

Definition 4.7. Forking *and* commuting diagrams *for a transformation t are meta-rewriting rules for some $r \subseteq \mathsf{ev}$, $t' \subseteq t$ and f being relations on processes.*

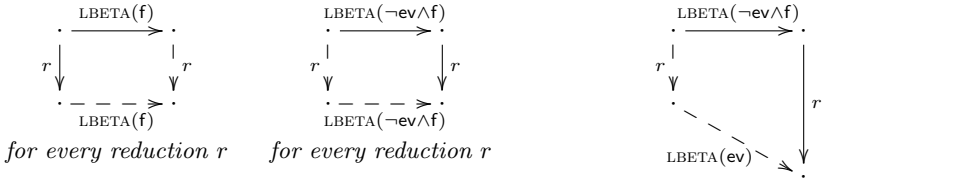
forking diagram commuting diagram

A set of forking diagrams $\{\xleftarrow{r_1} \cdot \xrightarrow{t_1} \rightsquigarrow \xrightarrow{f_1}, \ldots, \xleftarrow{r_n} \cdot \xrightarrow{t_n} \rightsquigarrow \xrightarrow{f_n}\}$ *is* complete *iff for every reduction sequence* $p_1 \xleftarrow{\mathsf{ev}} p_2 \xrightarrow{t} p_3$ *there exists a forking diagram* $r_i^{-1} \circ t_i \rightsquigarrow f_i$ *with* $(p_1, p_2) \in r_i^{-1}$, $(p_2, p_3) \in t_i$ *and* $(p_1, p_3) \in f_i$.

A set of commuting diagrams $\{\xrightarrow{t_1} \cdot \xrightarrow{r_1} \rightsquigarrow \xrightarrow{f_1}, \ldots, \xrightarrow{t_n} \cdot \xrightarrow{r_n} \rightsquigarrow \xrightarrow{f_n}\}$ *is* complete *iff for every reduction sequence* $p_1 \xrightarrow{t} p_2 \xrightarrow{\mathsf{ev}} p_3$ *there exists a commuting diagram* $t_i \circ r_i \rightsquigarrow f_i$ *with* $(p_1, p_2) \in t_i, (p_2, p_3) \in r_i$ *and* $(p_1, p_3) \in f_i$.

Lemma 4.8. *Let p_1, p_2 be two configurations with* $p_1 \xrightarrow{\text{LBETA}(\neg\mathsf{ev}\wedge\mathsf{a})} p_2$. *Then p_1 is successful iff p_2 is successful.*

Lemma 4.9. *A complete set of forking diagrams for* LBETA(f) *and a complete set of commuting diagrams for* LBETA($\neg$ev$\wedge$f) *are:*



for every reduction r *for every reduction r*

for $r \in \{\text{LBETA}(\mathsf{ev}), \text{LAZY.TRIGGER}(\mathsf{ev})\}$

Proposition 4.10. LBETA(f) *is a correct transformation.*

Proof. From Lemma 4.6 which is applicable by Lemmas 4.8 and 4.9 and from the fact that $\text{LBETA}(\mathsf{ev}) \circ \mathsf{ev} \subseteq \mathsf{ev}^*$. □

Theorem 4.11. GC *is a correct program transformation.*

Proof. This follows by Lemma 4.6, since GC has no influence on reduction sequences, i.e. $\mathsf{ev}^{-1} \circ \text{GC} \subseteq \text{GC} \circ \mathsf{ev}^{-1}$ and $\text{GC} \circ \mathsf{ev} \subseteq \mathsf{ev} \circ \text{GC}$ and since the conditions ensure that there is no interference with the success of processes. □

In the same way it follows that:

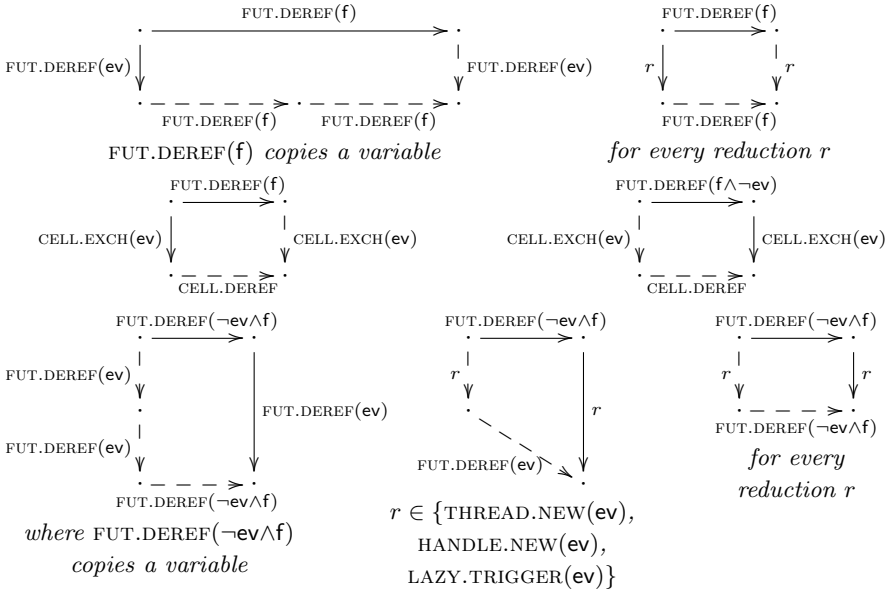
Theorem 4.12. DET.EXCH *is a correct program transformation.*

Lemma 4.13. *Let* p_1, p_2 *be processes with* $p_1 \xrightarrow{\text{FUT.DEREF}(\neg\mathsf{ev}\wedge\mathsf{a})} p_2$ *or* $p_1 \xrightarrow{\text{CELL.DEREF}} p_2$, *then p_1 is successful iff p_2 is successful.*

Lemma 4.14. *The forking and commuting diagrams for* CELL.DEREF *can be read off the following diagrams:*

for every reduction r

Lemma 4.15. *A complete set of forking (commuting, resp.) diagrams for* FUT.DEREF(f) *(*FUT.DEREF(¬ev∧f)*, resp.) can be read off the following diagrams:*



FUT.DEREF(f) *copies a variable*

for every reduction r

where FUT.DEREF(¬ev∧f) *copies a variable*

$r \in \{$THREAD.NEW(ev), HANDLE.NEW(ev), LAZY.TRIGGER(ev)$\}$

for every reduction r

Proposition 4.16. FUT.DEREF(f) *and* CELL.DEREF *are correct transformations.*

Proof. From Lemma 4.6 which is applicable by combining the diagrams of Lemmas 4.14, 4.13 and 4.15 and the fact that FUT.DEREF(ev) $\circ$ ev $\subseteq$ ev*. □

4.3 Correctness of FUT.DEREF(d) and β-CBV(a)

We strengthen this result for FUT.DEREF(f) in order to prove the correctness of FUT.DEREF(d) wrt. suitable measures on terms and sequences of reductions.

Definition 4.17. *Let p be a process with $p \equiv x_1 \Leftarrow x_2 \mid x_2 \Leftarrow x_3 \mid \ldots x_n \Leftarrow t \mid p'$ where t is not a variable and the chain of variables is maximal. Then the measure* $\mathrm{cl} : \mathit{Var} \to \mathbb{N}_0$ *is defined as* $\mathrm{cl}(x_1) := n$. *If the chain contains a chain of cyclic thread components $x_i \Leftarrow x_{i+1} \mid \ldots \mid x_j \Leftarrow x_i$ then* $\mathrm{cl}(x_1)$ *is undefined. The measure* $\#\mathrm{var}_\mathsf{f} : \mathit{Process} \to \mathbb{N}_0$ *is defined as follows: Let p be a process, then* $\#\mathrm{var}_\mathsf{f}(p)$ *is the sum of* $\mathrm{cl}(x)$ *of all occurrences of variables x in p, where the occurrence of x is inside a flat context.*

Let $R = p_0 \xrightarrow{\mathsf{ev}} p_1 \ldots \xrightarrow{\mathsf{ev}} p_n$ *be a sequence of reductions. Then* $\mathrm{rl}(R)$ *is the number of reductions of* R*, i.e.* $\mathrm{rl}(R) = n$*, and* $\mathrm{rl}_{\neg \text{FUT.DEREF}(\mathsf{ev})}(R)$ *is the number of reductions* r *of* R *with* $r \not\subseteq \text{FUT.DEREF}(\mathsf{ev})$*.*

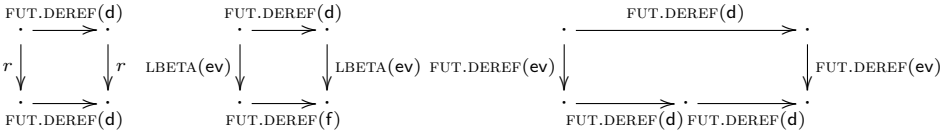
Lemma 4.18. *Let* p, p' *be two processes with* $p \xrightarrow{\text{FUT.DEREF}(\mathsf{f})} p'$*.*

i) If there exists $R \in \mathrm{Suc}(p)$ *(*$R \in \mathrm{Div}(p)$*, resp.) then there exists* $R' \in \mathrm{Suc}(p')$ *(*$R' \in \mathrm{Div}(p)$*, resp.) with* $\mathrm{rl}(R') \leq \mathrm{rl}(R)$*.*
ii) If there exists $R' \in \mathrm{Suc}(p')$ *(*$R' \in \mathrm{Div}(p')$*, resp.) then there exists* $R \in \mathrm{Suc}(p)$ *(*$R \in \mathrm{Div}(p)$*, resp.) with* $\mathrm{rl}_{\neg \text{FUT.DEREF}(\mathsf{ev})}(R) \leq \mathrm{rl}_{\neg \text{FUT.DEREF}(\mathsf{ev})}(R')$*.*

Proof. Follows by inspecting the forking and commuting diagrams for FUT.DEREF(f) that are used for the construction of the reduction sequences. □

Lemma 4.19. *If* p *is a process without cyclic chains of threads, then every* FUT.DEREF(d) *transformation preserves the measure* $\#\mathrm{var}_{\mathsf{f}}$*.*

Lemma 4.20. *Complete sets of forking and commuting diagrams for* FUT.DEREF(d) *can be read off the following diagrams for all reductions* r*:*



Proposition 4.21. *The transformation* FUT.DEREF(d) *is correct.*

Proof. Let $p_1 \xrightarrow{\text{FUT.DEREF}(\mathsf{d})} p_2$. We split the proof into four parts:

$\mathbf{p_1}{\downarrow} \Rightarrow \mathbf{p_2}{\downarrow}$: Let $R \in \mathrm{Suc}(p_1)$. We show by induction on $l = \mathrm{rl}(R)$ that there also exists $R' \in \mathrm{Suc}(p_2)$ with length $\leq l$.

Lemma 4.13 implies that the base case holds. Now let $l > 0$. Then we can apply one of the forking diagrams of Lemma 4.20 to a suffix of the sequence $\xleftarrow{R} p_1 \xrightarrow{\text{FUT.DEREF}(\mathsf{d})} p_2$ and then use the induction hypothesis. For the second diagram of Lemma 4.20 we apply Lemma 4.18, and for the last diagram we apply the induction hypothesis twice.

$\mathbf{p_2} \downarrow \Rightarrow \mathbf{p_1} \downarrow$: We use the (lexicographically ordered) measure (μ_1, μ_2) on reduction sequences of the form $p_1 \xrightarrow{\text{FUT.DEREF}(\mathsf{a})} p_2 \xrightarrow{R}$ with $R \in \mathrm{Suc}(p_2)$, $\mu_1 = \mathrm{rl}_{\neg \text{FUT.DEREF}(\mathsf{ev})}(R)$, and $\mu_2 = \#\mathrm{var}_{\mathsf{f}}(p_2)$. Note that μ_2 is defined, since by Lemma 3.1 the corresponding process does not contain a cyclic chain of threads.

Let $R \in \mathrm{Suc}(p_2)$. We show by induction on (μ_1, μ_2) that there exists $R' \in \mathrm{Suc}(p_1)$ with $\mathrm{rl}_{\neg \text{FUT.DEREF}(\mathsf{ev})}(R') \leq \mathrm{rl}_{\neg \text{FUT.DEREF}(\mathsf{ev})}(R)$.

For the base case let $(\mu_1, \mu_2) = (0, 0)$. Then Lemma 4.19 implies that R must be empty. Hence, p_2 is a successful process and Lemma 4.13 shows the claim. For the induction step let $(\mu_1, \mu_2) > (0, 0)$. We apply a commuting diagram from Lemma 4.20 to the sequence $p_1 \xrightarrow{\text{FUT.DEREF}(\mathsf{d})} p_2 \xrightarrow{R}$.

- If the first diagram is applicable, and the first reduction of R is a FUT.DEREF(ev) then μ_1 is unchanged, but μ_2 is strictly decreased. Otherwise μ_1 is strictly decreased. Hence we can apply the induction hypothesis.
- In case of the second diagram, μ_1 is strictly decreased and Lemma 4.18 shows the existence of R' with $\mathrm{rl}_{\neg\text{FUT.DEREF}(\mathsf{ev})}(R') \leq \mathrm{rl}_{\neg\text{FUT.DEREF}(\mathsf{ev})}(R)$.
- In case of the last diagram we can apply the induction hypothesis twice, since FUT.DEREF(ev) decreases the measure μ_2 and leaves μ_1 unchanged and a FUT.DEREF(d) transformation does not change μ_2 (see Lemma 4.19).

In any case, the constructed reduction sequence satisfies $\mathrm{rl}_{\neg\text{FUT.DEREF}(\mathsf{ev})}(R') \leq \mathrm{rl}_{\neg\text{FUT.DEREF}(\mathsf{ev})}(R)$.

$\mathbf{p_1}\uparrow \Rightarrow \mathbf{p_2}\uparrow$: This follows by induction on the length of a sequence $R \in \mathrm{Div}(p_1)$ and by using the forking diagrams. The base case follows from the previous case, $p_2\downarrow \Rightarrow p_1\downarrow$. The induction step is analogous to the first case of the proof.

$\mathbf{p_2}\uparrow \Rightarrow \mathbf{p_1}\uparrow$: This follows by induction on the measure (μ_1, μ_2) where $\mu_1 = \mathrm{rl}_{\neg\text{FUT.DEREF}(\mathsf{ev})}(R)$ with $R \in \mathrm{Div}(p_2)$ being a shortest sequence of reductions and $\mu_2 = \#\mathrm{var}_{\mathsf{f}}(p_2)$. Note that μ_2 may be undefined, but only for the last contractum of R, since R is a shortest sequence. Moreover, it is necessary to observe that FUT.DEREF(d) does not introduce cyclic chains of threads. The base case, i.e. $p_2\Uparrow$ follows from the first case of the proof, $p_1\downarrow \Rightarrow p_2\downarrow$. The induction step is analogous to the second case, using the commuting diagrams. □

Theorem 4.22. FUT.DEREF(a) *is a correct program transformation.*

Theorem 4.23. β-CBV(a) *is a correct program transformation.*

Proof. By the context lemma it suffices to show that β-CBV(f) is correct. In all flat contexts C, the transformation $C[(\lambda x.e)\ v] \xrightarrow{\beta\text{-CBV}(\mathsf{f})} C[e[v/x]]$ can be replaced by the sequence of transformations

$$\begin{aligned} C[(\lambda x.e)\, v] &\xrightarrow{\text{LBETA}(\mathsf{f})} (\nu x)\ C[e] \mid x{\Leftarrow}v \\ &\xrightarrow{\text{FUT.DEREF}(\mathsf{a})}{}^{*} (\nu x)\ C[e[v/x]] \mid x{\Leftarrow}v \xrightarrow{\text{GC}} C[e[v/x]] \end{aligned}$$

Since we have shown that LBETA(f), FUT.DEREF(a) and GC are correct in Theorems 4.11 and 4.22 and Proposition 4.10, respectively, the result follows. □

Theorem 4.24. *Path compression,* $(\nu y)(x{\Leftarrow}y \mid y{\Leftarrow}v) \to x{\Leftarrow}v$ *where* $y \notin \mathsf{fv}(v)$*, is correct.*

The theorems also imply that the following equivalence holds, which is not covered by the congruence property: $v_1 \sim v_2 \implies p \mid x\,\mathsf{c}\,v_1 \sim p \mid x\,\mathsf{c}\,v_2$, which follows from the equivalence $p \mid ((\nu y)(x\,\mathsf{c}\,y \mid y{\Leftarrow}v)) \sim p \mid x\,\mathsf{c}\,v$.

5 Conclusions and Outlook

We have presented an observational equivalence for $\lambda(\mathsf{fut})$ programs, which allows us to reason about the correctness of transformations of stateful and concurrent computations, as found in the Alice ML core language [16, 12]. Specifically,

we have proved correctness of partial evaluation with respect to this semantics. Equivalences like garbage collection and path compression are interesting, as they open the possibility of proving the actual Alice ML garbage collector correct. More generally, the framework developed in this paper can serve as the foundation for static analyses of higher-order concurrent languages like Alice ML.

The main tools to derive equivalences are a context lemma, and finding certain (complete sets of) rewrite rules on reduction sequences, called commuting and forking diagrams. The latter are adapted from [8, 17]; although the details differ, this technique appears both flexible and robust.

Future Work. We plan to investigate static analyses for $\lambda(\mathsf{fut})$, e.g. an adaptation of the calculus where touch optimization can be investigated [6]. Applying the correctness-criterion of must- and may-convergence to optimizations of the reduction strategy also deserves further work.

References

1. Henry Baker and Carl Hewitt. The incremental garbage collection of processes. *ACM Sigplan Notices*, 12:55–59, August 1977.
2. Arnaud Carayol, Daniel Hirschkoff, and Davide Sangiorgi. On the representation of McCarthy's amb in the pi-calculus. *Theor. Comp. Sci.*, 330(3):439–473, 2005.
3. Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Developing Applications With Objective Caml*. O'Reilly, 2000.
4. Sylvain Conchon and Fabrice Le Fessant. Jocaml: Mobile agents for Objective-Caml. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, 1999.
5. William Ferreira, Matthew Hennessy, and Alan Jeffrey. A theory of weak bisimulation for Core CML. *Journal of Functional Programming*, 8(5):447–491, 1998.
6. Cormac Flanagan and Matthias Felleisen. The semantics of future and an application. *Journal of Functional Programming*, 9(1):1–31, 1999.
7. Alan Jeffrey and Julian Rathke. A theory of bisimulation for a fragment of concurrent ML with local names. *Theor. Comp. Sci.*, 323(1-3):1–48, 2004.
8. Arne Kutzner and Manfred Schmidt-Schauß. A nondeterministic call-by-need lambda calculus. In *ICFP 1998*, pages 324–335. ACM Press, 1998.
9. Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, May 1999.
10. Robin Milner, Mads Tofte, Robert Harper, and David B. MacQueen. *The Standard ML Programming Language (Revised)*. MIT Press, 1997.
11. A. K. Moran. *Call-by-name, Call-by-need, and McCarthy's Amb*. PhD thesis, Gothenburg, Sweden, 1998.
12. Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A concurrent lambda calculus with futures. *Theor. Comp. Sci.*, 2006.
13. Andrew M. Pitts. Operational semantics and program equivalence. In J. T. O'Donnell, editor, *Applied Semantics*, volume 2395 of *Lecture Notes in Computer Science*, pages 378–412. Springer-Verlag, 2002.
14. Gordon D. Plotkin. A powerdomain construction. *SIAM Journal of Computing*, 5(3):452–487, 1976.
15. Andreas Rossberg. The missing link: dynamic components for ML. In *11th Int. Conf. on Functional Programming*, pages 99–110, 2006.

16. Andreas Rossberg, Didier Le Botlan, Guido Tack, Thorsten Brunklaus, and Gert Smolka. *Alice Through the Looking Glass*, volume 5 of TFP, pages 79–96. 2006.
17. David Sabel and Manfred Schmidt-Schauß. A call-by-need lambda-calculus with locally bottom-avoiding choice: Context lemma and correctness of transformations. Frank Report 24, Institut für Informatik, J.W. Goethe-Universität Frankfurt, 2006.
18. Davide Sangiorgi. The lazy lambda calculus in a concurrency scenario. *Information and Computation*, 111:120–153, 1994.
19. Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.
20. Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, volume 1000 of *LNCS*, pages 324–343. Springer, 1995.

A An Inference System for Well-Formedness

Fig. 6 presents an inference system for well-formed processes. A judgment $X \vdash p : Y$ means that Y is the set of process variables introduced by p and X is (a superset of) the set of all other free variables in p.[1] The rules ensure that all process variables are introduced exactly once. p is well-formed iff there are sets of variables X, Y such that $X \vdash p : Y$.

$$\frac{X \uplus Y_2 \vdash p_1 : Y_1 \qquad X \uplus Y_1 \vdash p_2 : Y_2}{X \vdash p_1 \mid p_2 : Y_1 \uplus Y_2} \qquad \frac{X \uplus \{x\} \supseteq \mathsf{fv}(e)}{X \vdash x \Leftarrow e : \{x\}}$$

$$\frac{X \uplus \{x\} \supseteq \mathsf{fv}(e)}{X \vdash x \overset{susp}{\Longleftarrow} e : \{x\}}$$

$$\frac{X \uplus \{x\} \supseteq \mathsf{fv}(v)}{X \vdash x \,\mathsf{c}\, v : \{x\}} \qquad \frac{x, y \notin X}{X \vdash y \,\mathsf{h}\, x : \{x, y\}}$$

$$\frac{X \vdash p : Y}{X \vdash (\nu x)p : Y \setminus \{x\}} \qquad \frac{y \notin X}{X \vdash y \,\mathsf{h}\, \bullet : \{y\}}$$

Fig. 6. Well-formedness

Lemma A.1. *Well-formedness is preserved by reduction.*

Proof. One first proves that well-formedness is closed under structural congruence and replacement of well-formed subconfigurations. More precisely, $X \vdash p : Y$ and $p \equiv p'$ implies $X \vdash p' : Y$, and for all process EC's D, if $X \vdash D[p] : Y$ then there exists X', Y' such that $X' \vdash p : Y'$ and $X \vdash D[p'] : Y$ for all $X' \vdash p' : Y'$. It is also not hard to see that if $X \vdash p : Y$ and Z is disjoint Y, then $X \cup Z \vdash p : Y$. Also note that for all $\tilde{E}$ and e, $\mathsf{fv}(\tilde{E}[e]) \supseteq \mathsf{fv}(e)$ (since in ECs the holes are not in the scope of binders), and for all e', one has $\mathsf{fv}(\tilde{E}[e']) \subseteq \mathsf{fv}(\tilde{E}[e]) \cup \mathsf{fv}(e')$.

Suppose $X \vdash p : Y$ for some X, Y, and $p \to p'$. The proof that $X \vdash p' : Y$ is now by a case distinction on the axioms and rules in Fig. 4. We only consider the case of (LBETA(ev)), the other cases are similar: By assumption, $X \vdash E[(\lambda z.e)\, v] : Y$ where $E = x \Leftarrow \tilde{E}$, so that $\mathsf{fv}(\tilde{E}[(\lambda z.e)\, v]) \subseteq X \uplus \{x\}$ and $Y = \{x\}$ by the well-formedness rules. By the above observations, $\mathsf{fv}((\lambda z.e)\, v) \subseteq X \uplus \{x\}$ which yields $\mathsf{fv}(\tilde{E}[e]) \subseteq X \uplus \{x\} \uplus \{z\}$ since, by the distinct variable convention, we may assume that z is fresh. Thus, $X \uplus \{z\} \vdash E[e] : \{x\}$. Moreover, $\mathsf{fv}(v) \subseteq X \uplus \{x\} \uplus \{z\}$ so that $X \uplus \{x\} \vdash z \Leftarrow v : \{z\}$. Using the well-formedness rule for parallel composition, this entails $X \vdash E[e] \mid z \Leftarrow v : \{x, z\}$ from which $X \vdash p' : Y$ follows by an application of the well-formedness rule for new name restriction. □

[1] The judgements are like the typing judgments from [12] but without type information.

B Prenex-Context Lemma for Processes

This section proves a context lemma for processes, mainly that we do not have to check all possibilities of ν-nesting.

A process context D is called *prenex*, iff it is of the form $(\nu x_1)\dots(\nu x_n)D'$ such that D' does not contain further ν-binders. A prenex process context D is called *ν-closed* iff every process variable in D is bound by a top-level ν-binder, and a process context D is called *ν-open (open)*, iff there are no ν-binders.

Lemma B.1. *Let p_1, p_2 be processes. Then $p_1 \sim p_2$ iff $p_1\rho \sim p_2\rho$ where ρ is a renaming of bound process variables that does not provoke any name clashes.*

Lemma B.2. *Let p_1, p_2 be processes. Then $p_1 \leq p_2$ iff all prenex process contexts D satisfy $D[p_1]\downarrow \Rightarrow D[p_2]\downarrow$ and $D[p_1]\Downarrow \Rightarrow D[p_2]\Downarrow$.*

Proof. We prove only the nontrivial direction. Assume the lemma is false: I.e. the condition for prenex process contexts holds, and there is a process context D, and p_1, p_2, such that $D[p_1]\downarrow$ but not $D[p_2]\downarrow$, or $D[p_1]\downarrow, D[p_2]\downarrow, D[p_1]\Downarrow$, but not $D[p_2]\Downarrow$. We show that in this case there is also a prenex process context D^{st} with this property. We show how to move the ν-binders in D to the toplevel. In the case $D = (\nu x)p_3 \mid D'[\,]$, we can also use $D'' = (\nu x')p_3[x'/x] \mid D'[\,]$, if x occurs as a free process variable in p_1, p_2, and x' is a fresh variable. Then the may- and must-convergence behavior of $D''[p_1]$ and $D[p_1]$ is equivalent, as well as that of $D''[p_2]$ and $D[p_2]$, by Lemma B.1. The second case is $D = (\nu x)D'[\,] \mid p_3$, where x is contained in p_3 and also in p_1 or p_2. In this case we obtain a new counterexample with $D'' = (\nu x')D'[x'/x][\,] \mid p_3$ and $p_1' = p_1[x'/x]$, $p_2' = p_2[x'/x]$. Now we can move ν upwards, obtaining a new counterexample with $D''' = (\nu x')(D'[x'/x][\,] \mid p_3)$.

The other cases of moving ν to the top are trivial. Finally, we obtain a counterexample to the assumed condition, which is impossible. Hence the lemma holds. □

Now we show that we can further specialize the test:

Lemma B.3. *Let p_1, p_2 be processes. Then $p_1 \leq p_2$ iff all prenex ν-closed process contexts D satisfy $D[p_1]\downarrow \Rightarrow D[p_2]\downarrow$ and $D[p_1]\Downarrow \Rightarrow D[p_2]\Downarrow$.*

Proof. If we proceed as in the previous proof, it is sufficient to argue that the may- and must-convergence behavior does not change if a top-level ν-binder is added. This is obvious, since reduction under ν-binders is always possible. □

The same reasoning shows that also the following holds:

Lemma B.4. *Let p_1, p_2 be processes. Then $p_1 \leq p_2$ iff all prenex ν-open process contexts D satisfy $D[p_1]\downarrow \Rightarrow D[p_2]\downarrow$ and $D[p_1]\Downarrow \Rightarrow D[p_2]\Downarrow$.*

In summary, these results show that we can more or less ignore the top-level ν-binders in the process contexts in further reasoning.

C Context Lemma for Expressions

The context lemma for expressions says that ECs provide enough observations to distinguish inequivalent expressions. It talks about the relations

$$e \leq^{\mathsf{ev}}_{\downarrow} e' \text{ iff } \forall E \forall D : D[E[e]]\downarrow \Rightarrow D[E[e']]\downarrow$$
$$e \leq^{\mathsf{ev}}_{\Downarrow} e' \text{ iff } \forall E \forall D : D[E[e]]\Downarrow \Rightarrow D[E[e']]\Downarrow$$

Proposition C.1 (Context Lemma for $\leq_\downarrow$). *For all expressions e_1, e_2:*

$$e_1 \leq^{\mathsf{ev}}_{\downarrow} e_2 \Rightarrow e_1 \leq_\downarrow e_2$$

This will follow from Lemma C.4. In a first step, we have to generalize the context lemma for expressions to multicontexts, which may have more than one hole, or none at all. A *multicontext* M with n holes is a process that permits additional constants $[]_1, \ldots, []_n$ for marking holes in expression positions, each of which occurs exactly once. We write $M[e_1, \ldots, e_n]$ for the process obtained by replacing $[]_i$ by e_i for all $1 \leq i \leq n$. Note that e.g. the terms $D[C_1[[]_1] \mid \ldots \mid C_n[[]_n]]$ are multicontexts with n holes. For instance, if M is $z \,\mathsf{c}\, \lambda x.([]_1\ []_2) \mid y \Leftarrow []_3$ then $M[x, y, z]$ becomes $z \,\mathsf{c}\, \lambda x.x\ y \mid y \Leftarrow z$ We assume $[] = []_1$ so that standard contexts C become multicontexts. The i-th hole of a multicontext M is *in EC position* if $M[e_1, \ldots, e_{i-1}, [], e_{i+1}, \ldots, e_n]$ is a (process) EC of the form $D[E]$ for some expressions $e_1, \ldots, e_n$.

Lemma C.2. *If the i-th hole of M is in EC position then there exists an index j such that $M[e_1, \ldots, e_{j-1}, [\,], e_{j+1}, \ldots, e_n]$ is an EC for all expressions $e_1, \ldots, e_n$.*

Proof. We prove the corresponding property for *expression* multicontexts, by an induction on such contexts $\tilde{M}$. In the case where $n \leq 1$ the proposition clearly holds. So suppose $\tilde{M}$ has at least two holes. Then either $\tilde{M}$ is of the form $\lambda x.\tilde{M}_1$ or $\tilde{M}_1\,\tilde{M}_2$ or $\mathbf{exch}(\tilde{M}_1, \tilde{M}_2)$. We may assume without loss of generality that for some $1 \leq k \leq n$, $\tilde{M}_1$ is a multicontext over holes $[]_1, \ldots, []_k$, and $\tilde{M}_2$ is a multicontext over $[]_{k+1}, \ldots, []_n$; otherwise we rename the holes accordingly.

- The case $\lambda x.\tilde{M}_1$ is not possible, since every instantiation of $n-1$ holes of M_1 yields a deep context. In particular, this cannot be an EC.
- In the case $\tilde{M}_1\,\tilde{M}_2$, we distinguish two subcases: First, if there exist $e_1, \ldots, e_k$ and $1 \leq i \leq k$ such that $\tilde{M}_1[e_1, \ldots, e_{i-1}, [\,], e_{i+1}, \ldots, e_k]$ is an EC, then by induction hypothesis there exists $1 \leq j \leq k$ such that $\tilde{M}_1[e'_1, \ldots, e'_{j-1}, [\,], e'_{j+1}, \ldots, e'_k]$ is an EC for all $e'_1, \ldots, e'_k$. Therefore, by definition of ECs and assumption $\tilde{M} = \tilde{M}_1\,\tilde{M}_2$, $\tilde{M}[e'_1, \ldots, e'_{j-1}, [\,], e'_{j+1}, \ldots, e'_n]$ Êis an EC for all $e'_1, \ldots, e'_n$.
  Second, if for all $e_1, \ldots, e_k$ and $1 \leq i \leq k$, $\tilde{M}_1[e_1, \ldots, e_{i-1}, [\,], e_{i+1}, \ldots, e_k]$ is *not* an EC then for all $e'_1, \ldots, e'_k$, $\tilde{M}_1[e'_1, \ldots, e'_k] \in \mathit{Val}$, for otherwise there is no instantiation of $\tilde{M} = \tilde{M}_1\,\tilde{M}_2$ that yields an EC, contradicting the assumption. Moreover, by assumption we have that $\tilde{M}_2[e_{k+1}, \ldots, e_{i-1}, [], e_{i+1}, \ldots, e_n]$ is an EC, for some $e_{k+1}, \ldots, e_n$ and $k+1 \leq i \leq n$. By induction hypothesis and $\tilde{M} = \tilde{M}_1\,\tilde{M}_2$, there exists $k+1 \leq j \leq n$ such that $\tilde{M}[e'_1, \ldots, e'_{j-1}, [\,], e'_{j+1}, \ldots, e'_n]$ is an EC for all $e'_1, \ldots, e'_n$.

- The case $\mathbf{exch}(\tilde{M}_1, \tilde{M}_2)$ is similar.

The statement of the lemma follows from this result by an induction on the structure of (process) multicontexts M. □

A redex R in a multicontext M is an EC $D[E]$ or future EC $D[F]$ in M to which some reduction rule applies. If $\overline{e} = e_1, \ldots, e_n$ is a sequence and $I \in \{1, \ldots, n\}^*$ a sequence of indices of length m then we write $\overline{e}^I$ for the sequence $e_{i_1}, \ldots, e_{i_m}$ where $I = i_1, \ldots, i_m$. We write $[]_I$ for the sequence of hole markers $[]_{i_1}, \ldots, []_{i_m}$.

Lemma C.3. *Let M be a deep multicontext with n holes (i.e., all n holes are below abstractions) and R be a redex of M, such that there is a reduction $\xrightarrow{r}$ at R. Then there exists a multicontext M' with n' holes and a sequence of indices $I \in \{1, \ldots, n\}^*$ of length n', such that if $\overline{e}$ and $\overline{e}'$ are any two sequences of expressions of length n, there is a renaming ρ such that*

$$M[\overline{e}] \to p \text{ at } R \;\Rightarrow\; p \equiv M'[\overline{e\rho}^I] \;\wedge\; M[\overline{e}'] \to M'[\overline{e'\rho}^I] \text{ at } R$$

for all processes p.

Proof. We treat only the two cases of the rules LBETA(ev) and FUT.DEREF(ev). We start with the rule LBETA(ev). Let the redex R be $D[E]$ and the term filling this context be $(\lambda z.e)\ v$. Since M is deep, the holes of M must either lie inside D, E, e, or v. Let I_1, I_2, I_3, I_4 be the sequence of numbers of holes in D, E, e, and v respectively, from the left to the right. Then we have

$$\begin{aligned} M[\overline{e}] &\equiv D[\overline{e}^{I_1}, E[\overline{e}^{I_2}, (\lambda z.e[\overline{e}^{I_3}])\ v[\overline{e}^{I_4}]]] \\ p &\equiv D[\overline{e}^{I_1}, (\nu z)(E[\overline{e}^{I_2}, e[\overline{e}^{I_3}]] \mid z \Leftarrow v[\overline{e}^{I_4}])] \end{aligned}$$

We can thus define the multicontext M' by $D[[]_{I_1}, (\nu z)(E[[]_{I_2}, e[[]_{I_3}]] \mid z \Leftarrow v[[]_{I_4}])]$ and set the sequence I to $I = 1, \ldots, n = I_1, I_2, I_3, I_4$. The required renaming ρ is the identity.

Only the case of FUT.DEREF(ev) needs a proper renaming, since a value is copied and some parameters e_i occurring in this value may get copied, too. Let the redex R be $D[F]$ of M, x the future that is dereferenced, and v its value. Since M is deep, the holes of M must either lie inside D, F, or v. Let I_1, I_2, I_3 be the sequence of numbers of holes in D, F, and v respectively, from the left to the right. Then the following holds:

$$\begin{aligned} M[\overline{e}] &\equiv D[\overline{e}^{I_1}, F[\overline{e}^{I_2}, x] \mid x \Leftarrow v[\overline{e}^{I_3}]] \\ p &\equiv D[\overline{e}^{I_1}, F[\overline{e}^{I_2}, v[\overline{e}^{I_3}]] \mid x \Leftarrow v[\overline{e}^{I_3}]] \end{aligned}$$

In a subsequent renaming step the thread $x \Leftarrow v[\overline{e}^{I_3}]$ will be renamed in order to satisfy the distinct variable convention. The renamings can be partitioned into renamings that are local to $\overline{e}$ or renamings in v, such that no hole is in the scope of the renamed variable. The interesting cases are the renamings of bound variables in v such that a hole in v is in the scope of such a variable. Since the

distinct variable convention was assumed to hold before the reduction, we can write ρ for the whole renaming (assuming that for non-renamed parts we have the identity). This means that we have renamed $v[\overline{e}^{I_3}]$ into $v\rho[\overline{e\rho}^{I_3}]$. Moreover, the renaming can be chosen such that its codomain is disjoint from the finitely many variables occuring in $\overline{e}'$. It is then clear that

$$M[\overline{e}] \to D[\overline{e'}^{I_1}, F[\overline{e'}^{I_2}, v[\overline{e'}^{I_3}]] \mid x \Leftarrow v\rho[\overline{e'\rho}^{I_3}]] \text{ at } R$$

We can now define the multicontext M' by $D[[]_{I_1}, F[[]_{I_2}, v[[]_{I'_3}]] \mid x \Leftarrow (v\rho)[[]_{I_3}]]$ where $I'_3 = n+1, \ldots, n+|I_3|$ and set the sequence I to $I = I_1, I_2, I_3, I'_3 = 1, \ldots, n+|I_3|$ where the holes with indices in I_3 get copied, and where $\overline{e\rho}$ and $\overline{e'\rho}$, resp., are used as parameters. □

Lemma C.4 (Generalized context lemma for may-convergence). *For $n \geq 0$ and $e_1, \ldots, e_n$ and $e'_1, \ldots, e'_n$ possibly empty sequences of expressions:*

$$(\forall 1 \leq i \leq n : \ e_i \leq^{\mathsf{ev}}_{\downarrow} e'_i) \quad \Rightarrow \quad \forall M : M[e_1, \ldots, e_n]{\downarrow} \Longrightarrow M[e'_1, \ldots, e'_n]{\downarrow}$$

Proof. Let $M[e_1, \ldots, e_n]\downarrow$. We use induction on the following lexicographic ordering of pairs (l, n), where

1. l is the length of a shortest succeeding sequence of reductions starting with $M[e_1, \ldots, e_n]$, and
2. n is the number of holes in M.

The claim holds for all pairs $(l, 0)$, since if M has no holes there is nothing to show. Now, let $(l, n) > (0, 0)$. For the induction step, we assume the claim holds for all pairs (l', n') that are strictly smaller than (l, n). We assume $\forall 1 \leq i \leq n : e_i \leq^{\mathsf{ev}}_{\downarrow} e'_i$ and let M be a multicontext with n holes such that $M[e_1, \ldots, e_n] \xrightarrow{l} p$ for some successful p. There are two cases:

- At least one hole of M is in EC position. Let this be the hole be in position $1 \leq i \leq n$. Let M_1 be the multicontext with $n-1$ holes defined by $M_1 \equiv M[[\,]_1, \ldots, [\,]_{i-1}, e_i, [\,]_{i+1}, \ldots, [\,]_n]$. Hence $M_1[e_1, \ldots, e_{i-1}, e_{i+1}, \ldots, e_n] \downarrow$ so that the induction hypothesis yields $M_1[e'_1, \ldots, e'_{i-1}, e'_{i+1}, \ldots, e'_n]\downarrow$. By Lemma C.2 there exists an EC $D'[E']$ such that $D'[E'] \equiv M[e'_1, \ldots, e'_{i-1}, [\,]_i, e'_{i+1}, \ldots, e'_n]$. Thus, $D'[\tilde{E}'[e_i]]\downarrow$ so that $D'[E'[e'_i]]\downarrow$ by assumption. The latter is $M[e'_1, \ldots, e'_n]\downarrow$.
- No hole of M is in EC position. If $l = 0$, then $M[e_1, \ldots, e_n]$ is successful, so $M[e'_1, \ldots, e'_n]$ is successful, too. Hence $M[e'_1, \ldots, e'_n]\downarrow$. If $l > 0$ then the first reduction step of $\xrightarrow{l}$ does also apply to $M[e'_1, \ldots, e'_n]$.
  Lemma C.3 shows that this reduction can only modify the context M, in that there exists a multicontext M' and a sequence of indices $I \in \{1, \ldots, n\}^*$ such that the result of the first reduction step of $M[\overline{e}]$ is $M'[\overline{e\rho}^I]$ where $\overline{e} = e_1, \ldots, e_n$, and $M[\overline{e'}] \to M'[\overline{e'\rho}^I]$ for the same renaming ρ, where $\overline{e'} = e'_1, \ldots, e'_n$ Note that M' may have more holes than M. We can apply the induction hypothesis to $M'[\overline{e\rho}^I]$ nevertheless, since there is a reduction of length $l-1$ to a successful process, and the hypotheses $e_i\rho \leq^{\mathsf{ev}}_{\downarrow} e'_i\rho$ still hold for all parameters of M'. □

Proposition C.5 (Context Lemma). *For all $e_1, e_2 \in Exp$:*

$$e_1 \leq^{\mathsf{ev}}_{\downarrow} e_2 \quad \textit{and} \quad e_1 \leq^{\mathsf{ev}}_{\Downarrow} e_2 \quad \Rightarrow \quad e_1 \leq e_2$$

Proof. The part for may-convergence follows from Proposition C.1. For the remaining part, we prove the claim that for all $n \geq 0$ and sequences $e_1, \ldots, e_n$, $e'_1, \ldots, e'_n$,

$$\forall 1 \leq i \leq n : \; e_i \leq^{\mathsf{ev}}_{\downarrow} e'_i \; \wedge \; e_i \leq^{\mathsf{ev}}_{\Downarrow} e'_i \; \Rightarrow \; (\forall M : M[e'_1, \ldots, e'_n]\uparrow \Rightarrow M[e_1, \ldots, e_n]\uparrow)$$

The lemma then follows using the remarks on the relations on may- and must-convergence and divergence in Section 3. We prove the claim by induction on lexicographically ordered pairs (l, n) where

1. l is the length of a shortest sequence of reductions starting with $M[e'_1, \ldots, e'_n]$ that ends in a process p with $p\Uparrow$, and
2. n is the number of holes in M.

If M has no holes there is nothing to show.
Now let $(l, n) > (0, 0)$. We analyze the two cases:

- At least one hole of M is in EC position. Then the same arguments as in the first part of the proof of Lemma C.4 show the claim.
- No hole of M is in EC position. If $l > 0$ then again the argumentation of part 2 of the proof of Lemma C.4 is used.
  The remaining case is $l = 0$, i.e., $M[e'_1, \ldots, e'_n]\Uparrow$. We have to show that $M[e'_1, \ldots, e'_n]\Uparrow \Rightarrow M[e_1, \ldots, e_n]\uparrow$. Using the relations between may- and must-convergence and must- and may-divergence, respectively, stated in Section 3, an equivalent claim is $M[e_1, \ldots, e_n]\Downarrow \Rightarrow M[e'_1, \ldots, e'_n]\downarrow$. Using the precondition and Lemma C.4 we have $M[e_1, \ldots, e_n]\downarrow \Rightarrow M[e'_1, \ldots, e'_n]\downarrow$. Since obviously $M[e_1, \ldots, e_n]\Downarrow$ implies $M[e_1, \ldots, e_n]\downarrow$, the claim follows. □

D Fairness

We show that may- and must-convergence do not change if reductions are restricted to be fair. Fairness is a property of reduction strategies, which we translate into a property of reduction sequences. It is necessary to speak of "the same redex" after some reductions. To this end we use a labelled reduction to identify the redex before and after reductions. Here by a redex we mean the expression in an E or F-context in the left process of the reduction rules in Fig. 4, with the exception of LAZY.TRIGGER(ev), where we mean the right (suspended) process.

Definition D.1. *A reduction sequence R starting from p is* fair *iff every redex is eventually reduced after a finite number of reductions of R. For a process p we define $p{\downarrow_{fair}}$ iff there is a fair reduction from p to a successful process, and $p{\Downarrow_{fair}}$ iff for every reduction $p \to^* p'$, we have $p'{\downarrow_{fair}}$.*

In general, different notions of fairness are conceivable. For instance, in *Weak and strong fairness in CCS* Costa & Stirling discuss several variants for Milner's CCS, which give rise to subtly differing views of fair reductions. Compared to CCS reduction, $\lambda(\mathsf{fut})$ processes are more well-behaved in that (1) there is no internal non-determinism within threads (cf. the CCS process $a.p+b.p'$), and (2) a reduction that is enabled in p but where the redex is not contracted in $p \to p'$ remains enabled in p' (unlike $(a.p_1 \mid \overline{a}.p_2 \mid a.p_3) \setminus a$ in CCS).

Since the definition of fair only excludes certain infinite reductions and the notion of may- and must-convergence is founded on finite reductions, the following is obvious:

Proposition D.2. *Let p be a process. Then $p{\downarrow} \Leftrightarrow p{\downarrow_{fair}}$ and $p{\Downarrow} \Leftrightarrow p{\Downarrow_{fair}}$.*

In contrast to other calculi, which may remove redexes without reduction, the reductions of $\lambda(\mathsf{fut})$ never remove redexes without reducing them:

Lemma D.3. *Let p be a process with $p{\downarrow}$, and let $p \to p'$ be a reduction. If p contains a redex that is not reduced in $p \to p'$, then that redex also appears in p'.*

The only case in the proof where it is not obvious that an unreduced redex is inherited is HANDLE.BIND(ev). In fact this is false whenever a handle can be used by two redexes, but then the process is easily seen to be must-divergent.

We give two examples of must-convergent processes with infinite reductions (but where the possibility of successful termination is preserved). In the first example the infinite reduction is fair. In contrast, the infinite reductions in the second example are not fair.

Example D.4. Let p be $z{\Leftarrow}y\,y \mid y{\Leftarrow}\lambda x.\mathit{choice}\,(\lambda\_.\mathbf{unit})\,(\lambda\_.x\,x)\,\mathbf{unit}$, with *choice* as in Example 2.1. Either a successful process is reached, if $(\lambda\_.\mathbf{unit})$ is selected, or the process $z{\Leftarrow}x\,x \mid x{\Leftarrow}y \mid y{\Leftarrow}\lambda x.\mathit{choice}\,(\lambda\_.\mathbf{unit})\,(\lambda\_.x\,x) \mid p'$ is reached after some reduction steps. This is a must-convergent process, and the infinite reductions are fair.

Example D.5. Let $K_{1!} = \lambda x.\lambda y.(x\,\mathbf{unit})$ and $K_2 = \lambda x\lambda y.y$. Let p the process $x_1{\Leftarrow}y\,y \mid z\,\mathsf{c}\,K_{1!} \mid x_2{\Leftarrow}\mathbf{exch}(z,K_2) \mid y{\Leftarrow}\lambda x.\mathbf{exch}(z,K_{1!})\,(\lambda\_.(x\,x))\,\mathbf{unit}$ which is must-convergent. There is a reduction to a successful process that first applies CELL.EXCH(ev) (so that the z-cell contains K_2), and subsequently reduces the x_1-thread to **unit**. In contrast, the unfair reduction never puts K_2 into the z-cell but always exchanges $K_{1!}$ with $K_{1!}$.

E Incorrectness of Transformations

Let $I = \lambda x.x$. We prove Lemma 4.2 by giving counter-examples for every rule. For notational simplicity, we omit the ν-binders.

THREAD.NEW(a): Let p_1 be the process $y{\Leftarrow}\lambda x.(\mathbf{thread}\,I)$. Thus, $p_1{\Downarrow}$ as it is already successful. Transforming it using THREAD.NEW(a) gives a process p_2, $y{\Leftarrow}\lambda x.z \mid z{\Leftarrow}(I\ z)$, which reduces to $y{\Leftarrow}\lambda x.z \mid z{\Leftarrow}u \mid u{\Leftarrow}z$, which is clearly must-divergent because of the cyclic subprocess, hence $p_2{\uparrow}$ follows.

HANDLE.NEW(a): Let p_1 be

$$y \Leftarrow \lambda x.\mathbf{handle}(\lambda u_1 \lambda u_2.u_2\,\mathbf{unit}) \mid x_1 \Leftarrow y\,\mathbf{unit} \mid x_2 \Leftarrow y\,\mathbf{unit}.$$

Reduction of p_1 creates two handles and then terminates with a successful process, thus $p_1\Downarrow$. In the case it is transformed, we obtain p_2,

$$y \Leftarrow \lambda x.((\lambda u_1 \lambda u_2.u_2\,\mathbf{unit})\,y_1\,y_2) \mid y_2\,\mathsf{h}\,y_1 \mid x_1 \Leftarrow y\,\mathbf{unit} \mid x_2 \Leftarrow y\,\mathbf{unit}$$

which will lead to a handle-error, i.e., p_2 is must-divergent.

HANDLE.BIND(a): Consider the process p_1,

$$m \Leftarrow \lambda x.y\,\mathbf{unit} \mid n \Leftarrow y\,\mathbf{unit} \mid y\,\mathsf{h}\,z$$

which is may- and must-convergent, whereas transformation results in p_2:

$$m \Leftarrow \lambda x.\mathbf{unit} \mid n \Leftarrow y\,\mathbf{unit} \mid y\,\mathsf{h}\,\bullet \mid z \Leftarrow \mathbf{unit}$$

Note that p_2 is not successful but also not reducible. Hence p_2 is must-divergent.

CELL.NEW(a): Applied within an abstraction, it is possible to share values, which are otherwise unshared, for instance let p_1, p_2 be the following processes:

$$\begin{aligned} p_1 \equiv\ & z \Leftarrow \lambda z.(\mathbf{cell}\,I) \\ \mid\ & x_1 \Leftarrow (\mathbf{exch}((z\,\mathbf{unit}), \mathbf{unit}))\,\mathbf{unit} \\ \mid\ & x_2 \Leftarrow (\mathbf{exch}((z\,\mathbf{unit}), \mathbf{unit}))\,\mathbf{unit} \end{aligned}$$

$$\begin{aligned} p_2 \equiv\ & z \Leftarrow \lambda z.w \\ \mid\ & w\,\mathsf{c}\,I \\ \mid\ & x_1 \Leftarrow (\mathbf{exch}((z\,\mathbf{unit}), \mathbf{unit}))\,\mathbf{unit} \\ \mid\ & x_2 \Leftarrow (\mathbf{exch}(((z\,\mathbf{unit}), \mathbf{unit}))\,\mathbf{unit} \end{aligned}$$

Process p_2 evolves from p_1 by applying CELL.NEW(a). We observe that $p_1\Downarrow$, since both **exch**-operations use their own cells and both will read the identity I. On the other hand, $p_2 \Uparrow$ since the **exch**-operations use the same cell, so that the thread performing the second exchange remains stuck with an application of the form **unit unit**.

CELL.EXCH(ev): The transformation is clearly not correct, since it can non-deterministically choose which exchange-operation to do first. The program $x \Leftarrow \mathbf{exch}(y, \mathbf{unit}) \mid z \Leftarrow \mathbf{exch}(y, \mathbf{unit}) \mid y\,\mathsf{c}\,x$ is may-convergent: after two reductions, the result is $x \Leftarrow \mathbf{unit} \mid z \Leftarrow x \mid y\,\mathsf{c}\,\mathbf{unit}$. Using the other possibility as transformation, a must-divergent program results: $x \Leftarrow x \mid z \Leftarrow \mathbf{exch}(y, \mathbf{unit}) \mid y\,\mathsf{c}\,\mathbf{unit}$.

LAZY.NEW(a): This rule is not correct inside abstractions, since there may be a sharing/desharing conflict: Let p_1 be

$$y\,\mathsf{c}\,I \mid w \Leftarrow \lambda x.\mathbf{lazy}(\lambda z.\mathbf{exch}(y, \mathbf{unit})) \mid w_2 \Leftarrow (w\,\mathbf{unit})\,(w\,\mathbf{unit})\,\mathbf{unit}$$

Applying LAZY.NEW(a) results in a process p_2,

$$\begin{array}{rl} p_2 \equiv & y \,\mathsf{c}\, I \\ \mid & w \Leftarrow \lambda x.w' \\ \mid & w' \overset{susp}{\Longleftarrow} (\lambda z.\mathbf{exch}(y, \mathbf{unit}))\, w' \\ \mid & w_2 \Leftarrow (w\,\mathbf{unit})\,(w\,\mathbf{unit})\,\mathbf{unit} \end{array}$$

Note that $p_2 \Uparrow$, since only one **exch**-operation is performed (reading the identity I), whereas p_1 does not converge, since two exchange-operations are performed, and thus one of these results in **unit**.

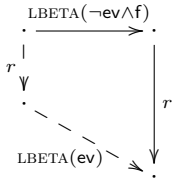
LAZY.TRIGGER(f): This transformation is not correct in arbitrary contexts, since it would force evaluation. An easy counterexample is $y \Leftarrow x \mid x \overset{susp}{\Longleftarrow} x$ which is convergent (it is successful), but becomes must-divergent after forcing the evaluation (because of the cyclic $x \Leftarrow x$).

F Examples for the Forking and Commuting Diagrams

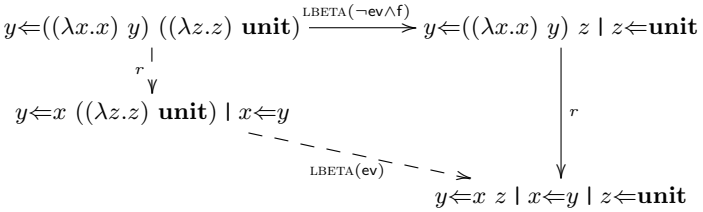
Lemmas 4.9, 4.15, and 4.20 follow by analyzing all overlappings of the corresponding transformation with reductions. In this section we give typical example cases for the non-trivial diagrams.

F.1 Diagrams for LBETA(¬ev∧f)

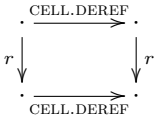
The non-trivial cases occur when an LBETA(¬ev∧f) becomes a reduction LBETA(ev), as expressed by the diagram

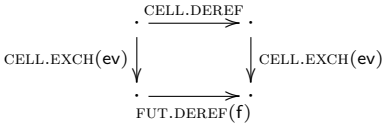


This case may occur if r is a LAZY.TRIGGER(ev) reduction, and if the redex of the transformation LBETA(¬ev∧f) is inside a lazy future that gets triggered. Another case is that r is an LBETA(ev) reduction, e.g.



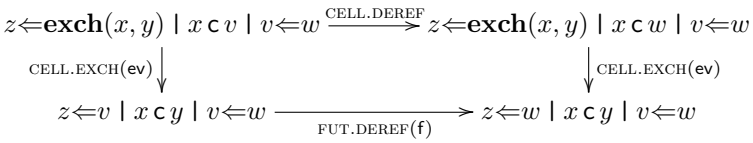
F.2 Diagrams for CELL.DEREF





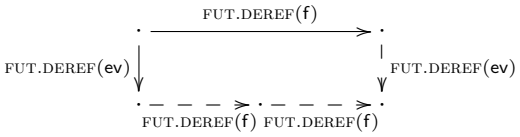
for every reduction r

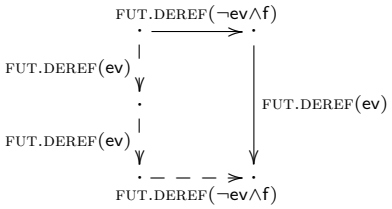
Cases for the first diagram are obvious. An example for the second diagram is:



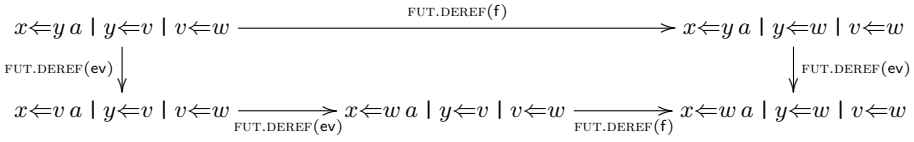
F.3 Diagrams for FUT.DEREF(f) and FUT.DEREF(¬ev∧f)

We show typical examples for the diagrams of Lemma 4.15.

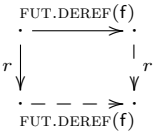




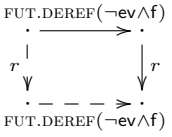
A typical example for both diagrams is:



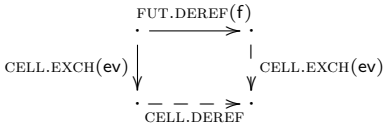
We now examine the following four diagrams:

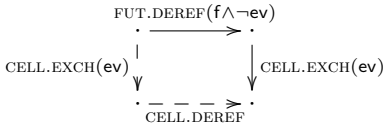


for every reduction r

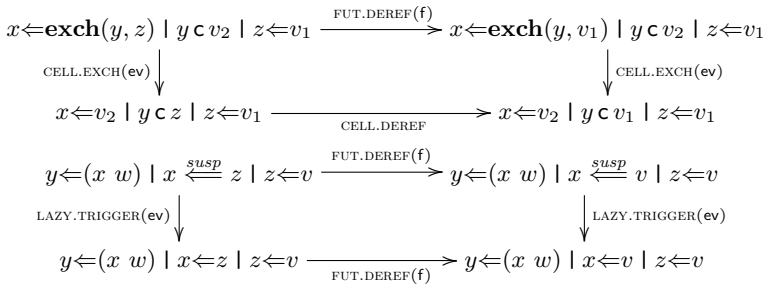


for every reduction r

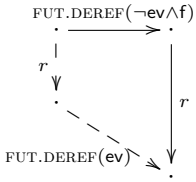




Beside simple commuting cases, where the transformation and the standard reduction do not influence each other, there are cases where the target of the dereferencing operation moves from a thread to a lazy thread or to a cell and vice versa. We show two examples:

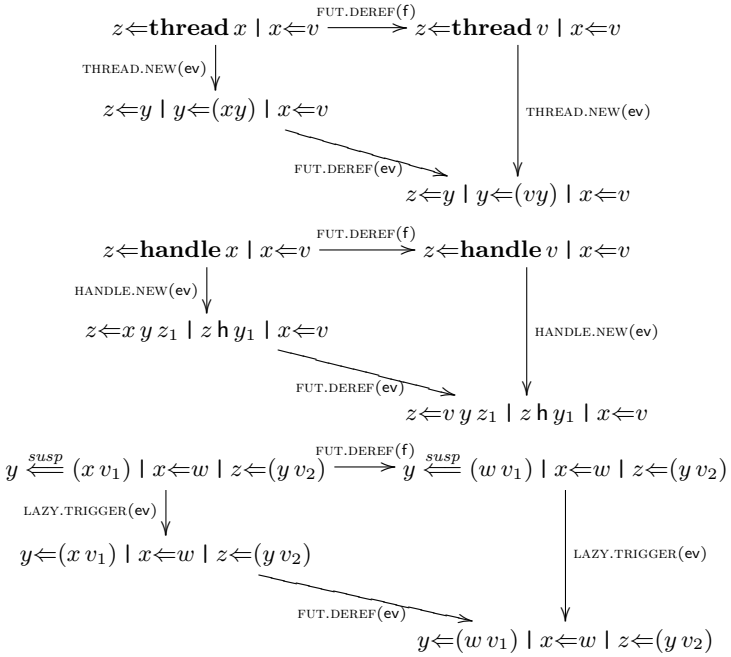


We now look at the diagram

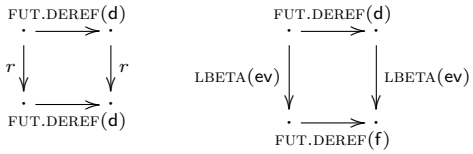


for $r \in \{\text{THREAD.NEW}(\mathsf{ev}), \text{HANDLE.NEW}(\mathsf{ev}), \text{LAZY.TRIGGER}(\mathsf{ev})\}$

We illustrate all three cases by examples:

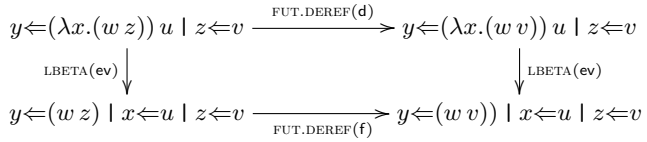


F.4 Diagrams for FUT.DEREF(d)

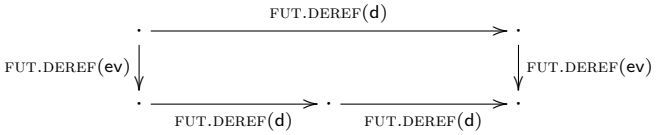


for every reduction r

The first diagram has the same special cases as the diagram for FUT.DEREF(f). The second diagram shows the only case where the target of a dereferencing operation is inside the body of an abstraction, but this is no longer the case after applying a standard reduction. An example for this case is:



The last diagram is:



An example for this case is: