An Abstract Machine for Concurrent Haskell with Futures

David Sabel

Computer Science Institute
Goethe-University Frankfurt am Main
sabel@ki.informatik.uni-frankfurt.de

Abstract: We show how Sestoft's abstract machine for lazy evaluation of purely functional programs can be extended to evaluate expressions of the calculus CHF – a process calculus that models Concurrent Haskell extended by imperative and implicit futures. The abstract machine is modularly constructed by first adding monadic IO-actions to the machine and then in a second step we add concurrency. Our main result is that the abstract machine coincides with the original operational semantics of CHF, w.r.t. may- and should-convergence.

1 Introduction

The process calculus *CHF* [SSS11a] is a model of the core language of Concurrent Haskell [PGF96, Pey01, PS09] but extended by implicit, concurrent futures which allow a declarative style of concurrent programming.

CHF is monomorphically typed and its syntax comprises (unlike the π-calculus [Mil99, SW01]) shared memory in form of Haskell's MVars, threads (i.e. futures) and heap bindings. Threads evaluate expressions which on the one hand may be monadic operations to create and access the MVars and to spawn new threads, and on the other hand are usual pure functional expressions extending the lambda calculus by data constructors, case-expressions, recursive let-expressions, as well as Haskell's `seq`-operator.

In [SSS11a] the operational semantics of *CHF* is defined by a small-step reduction as rewriting on processes. Program equivalence of processes and also expressions is given by a *contextual equivalence*: two programs are equal iff their observable behavior is indistinguishable even if the programs are used as a subprogram of any other program (i.e. if the programs are plugged into any arbitrary context). Besides observing whether a program *may* terminate (called *may-convergence*) contextual equivalence also tests whether a program never loses the ability to terminate (called *should-convergence*, or sometimes must-convergence, see e.g. [CHS05, NSSSS07, RV07, SSS08]). The classic notion of must-convergence additionally requires that all possible evaluations terminate. An advantage of using should-convergence is that it is invariant w.r.t. restricting the evaluator to fair scheduling (see e.g [SSS11a]), that contextual equivalence is closed w.r.t. a whole class of convergence predicates (see [SSS10]), and that inductive reasoning is possible.

In [SSS11a] contextual equivalence in *CHF* is deeply investigated and a lot of equiva-

lences are proved, and recently [SSS11b] shows that CHF is a conservative extension of its purely functional sublanguage, i.e. all equations that hold in the pure call-by-need lambda calculus also hold in the process calculus CHF. The obtained results show that the given operational semantics works well for (mathematically formal) reasoning. On the other hand the operational semantics is not easy to implement as an interpreter for CHF, since e.g. reduction contexts in [SSS11a] have a complex definition and reduction uses structural congruence of processes implicitly.

Hence, the motivation of this paper is to investigate an alternative operational semantics for CHF which can easily be implemented as an interpreter, i.e. we will develop an *abstract machine* to evaluate expressions and processes of CHF. As a starting point, we will use the abstract machine mark 1 introduced by Sestoft [Ses97] for call-by-need evaluation of pure functional programs (which implements the natural semantics given by [Lau93]).

Sestoft's machine mark 1 is a variant of the Krivine-machine which additionally implements sharing during evaluation (see [DF07]). The main components are a heap to model shared bindings, an expression which is evaluated, and a stack to efficiently store the current evaluation context. There are only few transition rules which perform the unwinding to find the next redex, perform reduction, or access and update shared bindings.

Variants of Sestoft's machine are well-used for several call-by-need lambda calculi to define the operational semantics, or to give an alternative description of the semantics, respectively. Some examples are [MSC99] for a call-by-need lambda calculus with erratic choice, [Mor98, Sab08] for call-by-need lambda calculi with McCarthy's amb-operator, [BFKT00] for specifying the semantics of Parallel Haskell, and [AHH$^+$05] for the semantics of functional-logic languages.

To construct an abstract machine for CHF we extend (a slightly modified variant of) Sestoft's machine (called $M1$) in two steps. The first extension (called $IOM1$) is to add the ability to perform monadic I/O-operations, i.e. we add storage (i.e. MVars), a further stack, and machine transitions to execute monadic actions to the machine $M1$. In a second step we extend the machine $IOM1$ by concurrency, i.e. we allow several threads and add transitions to spawn new threads. The concurrent machine is called $CIOM1$. A nice property of our construction is *modularity*, i.e. every extended machine reuses the already introduced transitions of the machine before. Thus $CIOM1$ is easy to implement, and indeed within a few hours we programmed a prototype of the machine in Haskell.

Albeit providing such a machine is an interesting result for itself, we also show that our machine is a *correct* implementation of the operational semantics of CHF: In Theorem 4.11 we show that may- and should-convergence defined by the rewriting semantics in [SSS11a] coincides with may- and should-convergence on the machine for every expression the machine starts with.

The structure of the paper is as follows: In Section 2 we briefly recall the calculus CHF together with some results on program equivalences in CHF which are required in later proofs. In Section 3 we introduce the abstract machine $CIOM1$ for CHF, where we develop the machine in three steps. In Section 4 we show that machine $CIOM1$ correctly implements the operational semantics of CHF. We conclude in Section 5. Not all proofs are included in the paper, but can be found in the technical report [Sab12].

2 The Process Calculus CHF

We recall the calculus CHF which models Concurrent Haskell extended by concurrent futures [SSS11a]. In Fig. 1(a) the syntax of *processes Proc* and expressions *Exp* is shown, where we assume that x, x_i, y, y_i are variables of some countably infinite set of variables.

Parallel composition $P_1 \mid P_2$ composes processes, and *name restriction* $\nu x.P$ restricts the scope of variable x to process P. A *concurrent thread* $x \Leftarrow e$ evaluates the expression e and then binds the result to the variable x. We also call variable x the *future* x. In a process there is usually one distinguished thread – the *main thread* – which is labeled with "main" (as notation we use $x \overset{\mathsf{main}}{\Longleftarrow} e$). *Bindings* $x = e$ represent global shared expressions. *MVars* are synchronizing variables, where $x\,\mathbf{m}\,e$ represents a *filled MVar* with content e, and $x\,\mathbf{m}-$ represents an *empty MVar*. In both cases we call x the *name of the MVar*. For a process P we say a variable x is an *introduced variable* if x is a future, a name of an MVar, or a left hand side of a binding. A process is *well-formed*, if there exists at most one main thread $x \overset{\mathsf{main}}{\Longleftarrow} e$ and the introduced variables are pairwise distinct.

We assume a finite set of *data constructors* c which is partitioned into sets, such that each set represents a type T. The constructors of a type T are ordered as $c_{T,1}, \ldots, c_{T,|T|}$, where $|T|$ is the number of constructors belonging to type T. We omit the index T, i in $c_{T,i}$ if it is clear from the context. Each constructor $c_{T,i}$ has a fixed arity $\mathrm{ar}(c_{T,i}) \geq 0$. We assume that there is a unit type $()$ with a single constant $()$ as constructor.

Besides the lambda calculus, expressions *Exp* (see Fig. 1(a)) comprise (fully-saturated) *constructor applications* $(c\ e_1 \ldots e_{\mathrm{ar}(c)})$, `case`*-expressions*, `seq`-expressions for sequential evaluation, `letrec`*-expressions* to express recursive shared bindings and monadic expressions *MExp* (described below). For `case`-expressions there is a $\texttt{case}_T$-construct for every type T which must have a `case`-alternative for every constructor of type T. We sometimes abbreviate the `case`-alternatives as $alts$. Variables in `case`-patterns $(c\ x_1 \ldots x_{\mathrm{ar}(c)})$ and bound variables in `letrec`-expressions must be pairwise distinct.

The monadic expression `return` e represents the monadic action which returns expression e, the binary operator $>\!\!>\!\!=$ combines monadic actions, the expression `future` e creates a concurrent thread evaluating the action e, the operation `newMVar` e creates an MVar filled with e, `takeMVar` x returns the content of MVar x, and `putMVar` $x\ e$ fills MVar x with e. `takeMVar` x blocks on an empty MVar, and `putMVar` $x\ e$ blocks on a filled MVar.

Example 2.1. *Futures allow a declarative programming style, since they allow implicit synchronization. Assume that* act_1, act_2 *perform computations resulting in numbers, and that we want to sum up both results when they are available, then we can use the action:*

$$\texttt{future}\ act_1 >\!\!>\!\!= \lambda res_1.\texttt{future}\ act_2 >\!\!>\!\!= \lambda res_2.\texttt{return}\ (res_1 + res_2)$$

Executing this action starts two concurrent futures for performing the actions act_1 *and* act_2*. The corresponding futures* res_1 *and* res_2 *are like pointers that will eventually point to the corresponding results. The futures are implicit, since there is no need to explicitly force the results of* res_1, res_2 *before computing the sum.*

Variables get bound by abstractions, `letrec`-expressions, `case`-alternatives, and by the

$P, P_i \in \mathit{Proc} ::= P_1 \mid P_2 \mid \nu x.P \mid x \Leftarrow e \mid x = e \mid x\,\mathbf{m}\,e \mid x\,\mathbf{m}-$

$e, e_i \in \mathit{Exp} ::= x \mid m \mid \lambda x.e \mid (e_1\ e_2) \mid c\,e_1 \ldots e_{\mathrm{ar}(c)} \mid \mathtt{seq}\ e_1\ e_2$
$\mid \mathtt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \mathtt{in}\ e$
$\mid \mathtt{case}_T\ e\ \mathtt{of}\ alt_{T,1} \ldots\ alt_{T,|T|}$ where $alt_{T,i} = (c_{T,i}\ x_1 \ldots x_{\mathrm{ar}(c_{T,i})} \to e_i)$

$m \in \mathit{MExp} ::= \mathtt{return}\,e \mid e_1 \gg\!= e_2 \mid \mathtt{future}\,e$
$\mid \mathtt{takeMVar}\,e \mid \mathtt{newMVar}\,e \mid \mathtt{putMVar}\,e_1\,e_2$

$\tau, \tau_i \in \mathit{Typ} ::= \mathtt{IO}\,\tau \mid (T\ \tau_1 \ldots \tau_n) \mid \mathtt{MVar}\,\tau \mid \tau_1 \to \tau_2$

(a) Syntax of Processes, Expressions, Monadic Expressions and Types

$\mathbb{D} \in \mathit{PC} ::= [\cdot] \mid \mathbb{D} \mid P \mid P \mid \mathbb{D} \mid \nu x.\mathbb{D}$ $\qquad$ $\mathbb{M} \in \mathit{MC} ::= [\cdot] \mid \mathbb{M} \gg\!= e$

$\mathbb{E} \in \mathit{EC} ::= [\cdot] \mid (\mathbb{E}\ e) \mid (\mathtt{case}\ \mathbb{E}\ \mathtt{of}\ alts) \mid (\mathtt{seq}\ \mathbb{E}\ e)$

$\mathbb{F} \in \mathit{FC} ::= \mathbb{E} \mid (\mathtt{takeMVar}\ \mathbb{E}) \mid (\mathtt{putMVar}\ \mathbb{E}\ e)$

$\mathbb{L} \in \mathit{LC} ::= x \Leftarrow \mathbb{M}[\mathbb{F}] \mid x \Leftarrow \mathbb{M}[\mathbb{F}[x_n]] \mid x_n = \mathbb{E}_n[x_{n-1}] \mid \ldots \mid x_2 = \mathbb{E}_2[x_1] \mid x_1 = \mathbb{E}_1$
where $\mathbb{E}_2, \ldots \mathbb{E}_n$ are not the empty context.

$\widehat{\mathbb{L}} \in \widehat{\mathit{LC}} ::= x \Leftarrow \mathbb{M}[\mathbb{F}] \mid x \Leftarrow \mathbb{M}[\mathbb{F}[x_n]] \mid x_n = \mathbb{E}_n[x_{n-1}] \mid \ldots \mid x_2 = \mathbb{E}_2[x_1] \mid x_1 = \mathbb{E}_1$
where $\mathbb{E}_1, \mathbb{E}_2, \ldots \mathbb{E}_n$ are not the empty context.

(b) Process-, Monadic-, Evaluation-, and Forcing-Contexts

Monadic Computations:

(lunit) $y \Leftarrow \mathbb{M}[\mathtt{return}\ e_1 \gg\!= e_2] \xrightarrow{\mathit{CHF}} y \Leftarrow \mathbb{M}[e_2\ e_1]$

(tmvar) $y \Leftarrow \mathbb{M}[\mathtt{takeMVar}\ x] \mid x\,\mathbf{m}\,e \xrightarrow{\mathit{CHF}} y \Leftarrow \mathbb{M}[\mathtt{return}\ e] \mid x\,\mathbf{m}-$

(pmvar) $y \Leftarrow \mathbb{M}[\mathtt{putMVar}\ x\ e] \mid x\,\mathbf{m}- \xrightarrow{\mathit{CHF}} y \Leftarrow \mathbb{M}[\mathtt{return}\ ()] \mid x\,\mathbf{m}\,e$

(nmvar) $y \Leftarrow \mathbb{M}[\mathtt{newMVar}\ e] \xrightarrow{\mathit{CHF}} \nu x.(y \Leftarrow \mathbb{M}[\mathtt{return}\ x] \mid x\,\mathbf{m}\,e)$

(fork) $y \Leftarrow \mathbb{M}[\mathtt{future}\ e] \xrightarrow{\mathit{CHF}} \nu z.(y \Leftarrow \mathbb{M}[\mathtt{return}\ z] \mid z \Leftarrow e)$
where z is fresh and the created thread is not the main thread

(unIO) $y \Leftarrow \mathtt{return}\ e \xrightarrow{\mathit{CHF}} y = e$ if the thread is not the main-thread

Functional Evaluation:

(cp) $\widehat{\mathbb{L}}[x] \mid x = v \xrightarrow{\mathit{CHF}} \widehat{\mathbb{L}}[v] \mid x = v$ $\qquad$ if v is an abstraction or a variable

(cpcx) $\widehat{\mathbb{L}}[x] \mid x = c\,e_1 \ldots e_n$ $\qquad$ if c is a constructor, or a monadic operator
$\xrightarrow{\mathit{CHF}} \nu y_1, \ldots y_n.(\widehat{\mathbb{L}}[c\,y_1 \ldots y_n] \mid x = c\,y_1 \ldots y_n \mid y_1 = e_1 \mid \ldots \mid y_n = e_n)$

(mkbinds) $\mathbb{L}[\mathtt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \mathtt{in}\ e]$
$\xrightarrow{\mathit{CHF}} \nu x_1, \ldots, x_n.(\mathbb{L}[e] \mid x_1 = e_1 \mid \ldots \mid x_n = e_n)$

(lbeta) $\mathbb{L}[((\lambda x.e_1)\ e_2)] \xrightarrow{\mathit{CHF}} \nu x.(\mathbb{L}[e_1] \mid x = e_2)$

(case) $\mathbb{L}[\mathtt{case}_T\ (c\,e_1\ \ldots\ e_n)\ \mathtt{of}\ \ldots (c\,y_1\ \ldots\ y_n \to e)\ldots]$
$\xrightarrow{\mathit{CHF}} \nu y_1, \ldots, y_n.(\mathbb{L}[e] \mid y_1 = e_1 \mid \ldots \mid y_n = e_n])$

(seq) $\mathbb{L}[(\mathtt{seq}\ v\ e)] \xrightarrow{\mathit{CHF}} \mathbb{L}[e]$ $\qquad$ if v is a functional value

Closure: If $P_1 \equiv \mathbb{D}[P_1']$, $P_2 \equiv \mathbb{D}[P_2']$, and $P_1' \xrightarrow{\mathit{CHF}} P_2'$ then $P_1 \xrightarrow{\mathit{CHF}} P_2$

(c) Standard Reduction Rules

Figure 1: The Calculus *CHF*

restriction $\nu x.P$. This induces a notion of free and bound variables. With $FV(P)$ ($FV(e)$, resp) we denote the free variables of process P (expression e, resp.) and with $=_\alpha$ we denote α-equivalence. We assume that the *distinct variable convention* holds, i.e. all free variables are distinct from bound variables, all bound variables are pairwise distinct, and reductions implicitly perform α-renaming to obey this convention. For processes *structural congruence* $\equiv$ is defined as the least congruence satisfying the equations $P_1 \mid P_2 \equiv P_2 \mid P_1$; $\nu x_1.\nu x_2.P \equiv \nu x_2.\nu x_1.P$; $(P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3)$; $P_1 \equiv P_2$, if $P_1 =_\alpha P_2$; and $(\nu x.P_1) \mid P_2 \equiv \nu x.(P_1 \mid P_2)$, if $x \notin FV(P_2)$.

For typing of processes and expressions *CHF* uses a monomorphic type system where data constructors and monadic operators are treated like "overloaded" polymorphic constants. The syntax of types *Typ* is shown in Fig. 1(a). $\texttt{IO}\,\tau$ means a monadic action with result type τ, $\texttt{MVar}\,\tau$ means an `MVar`-reference with content type τ, and $\tau_1 \to \tau_2$ is a function type. For a constructor c we let $\texttt{types}(c)$ be set of its monomorphic types. For simplicity we assume that every variable x has a fixed (built-in) type given by a global typing function Γ, i.e. $\Gamma(x)$ is the type of variable x. For space reasons we omit the typing rules of [SSS11a], but we use the notation $\Gamma \vdash P :: \texttt{wt}$ ($\Gamma \vdash e :: \tau$, resp.) meaning that (well-formed) process P can be well-typed (expression e can be well-typed with type τ, resp.) using the global typing function Γ. Special typing restrictions are that $x \Leftarrow e$ is well-typed, if $\Gamma \vdash e :: \texttt{IO}\,\tau$, and $\Gamma(x) = \tau$, and that the first argument of `seq` must not be an `IO`- or `MVar`-type, since otherwise the monad laws would not hold in *CHF* (and even not in Haskell, see [SSS11a]).

Operational Semantics and Program Equivalence The operational semantics of *CHF* (see [SSS11a]) is given by a small-step reduction which implements a call-by-need strategy. The definition requires several classes of contexts, which are shown in Fig. 1(b). For processes there are *process contexts PC*. For expressions, *monadic contexts MC* are used to find the first monadic action in a sequence of actions. For the evaluation of pure expressions, usual (call-by-name) *expression evaluation contexts EC* are used, and to enforce the evaluation of the (first) argument of the monadic operators `takeMVar` and `putMVar`, the class of *forcing contexts FC* is used. Since we follow a call-by-need strategy, we sometimes need to search a redex along a chain of bindings, which is expressed by the *LC*-contexts and as a special case by the $\widehat{LC}$-contexts.

Definition 2.2 (Call-by-Need Standard Reduction)**.** *A* functional value *is an abstraction or a constructor application, a* value *is a functional value or a monadic expression of MExp. The* call-by-need standard reduction $\xrightarrow{CHF}$ *is defined by the rules and the closure in Fig. 1(c). We assume that only well-formed processes are reducible. We also assume that successful processes (see below, Definition 2.3) are irreducible.*

The rules for functional evaluation include a sharing variant of β-reduction (rule (lbeta)), a rule for copying shared bindings into a needed position: For abstractions rule (cp) is used and for constructor applications rule (cpcx) shares the arguments before copying the constructor. The rules (case) and (seq) evaluate `case`- and `seq`-expressions, and the rule (mkbinds) moves `letrec`-bindings into the global set of shared bindings. For monadic computations the rule (lunit) applies the first monad law to proceed a sequence of monadic actions. The rules (nmvar), (tmvar), and (pmvar) handle the creation of and the access to

MVars where (tmvar) can only be performed on a filled MVar, and (pmvar) requires an empty MVar. The rule (fork) spawns a new concurrent thread, where the calling thread receives the name of the future as result. If a concurrent thread finishes its computation, then the result is shared as a global binding and the thread is removed (rule (unIO)).

For a reduction $\rightarrow$ (and also transitions and transformations) we denote with $\xrightarrow{+}$, $\xrightarrow{*}$ the transitive and the reflexive-transitive closure of $\rightarrow$, respectively. The notation $\xrightarrow{k}$ means a sequence of k $\rightarrow$-steps and $\xrightarrow{0\vee1}$ mean one or no reduction. We also sometimes attach a specific label to the arrow if we mean a specific reduction, and also write (CHF, a) for a CHF-standard reduction of kind a.

Contextual equivalence equates two processes P_1, P_2 if their observable behavior is indistinguishable if P_1 and P_2 are plugged into any process context. Thereby the usual observation is whether the evaluation of the process successfully terminates (called may-convergence). However, this observation is not sufficient in a concurrent setting, and thus we will observe may-convergence and a variant of must-convergence (called should-convergence, see also [RV07, SSS08, SSS11a]):

Definition 2.3. *A process P is* successful *iff it is well-formed and contains a main thread of the form $x \overset{\mathsf{main}}{\Longleftarrow} \texttt{return}\ e$. A process P* may-converges *(written as $P{\downarrow}$), iff it is well-formed and reduces to a successful process, i.e. $\exists P' : P \xrightarrow{CHF,*} P' \wedge P'$ is successful. If $P{\downarrow}$ does not hold, then P* must-diverges *written as $P{\Uparrow}$. A process P* should-converges *(written as $P{\Downarrow}$), iff it is well-formed and remains may-convergent under reduction, i.e. $\forall P' : P \xrightarrow{CHF,*} P' \implies P'{\downarrow}$. If P is not should-convergent, then we say P* may-diverges*, written as $P{\uparrow}$. For an expression $e :: \texttt{IO}\ \tau$ we write $e\chi$ for any $\chi \in \{{\downarrow}, {\Downarrow}, {\uparrow}, {\Uparrow}\}$ iff $P\chi$ where $P := x \overset{\mathsf{main}}{\Longleftarrow} e$ and $x \notin FV(e)$.*

Note that $P{\uparrow}$ iff there is a finite reduction sequence $P \xrightarrow{CHF,*} P'$ such that $P'{\Uparrow}$.

Definition 2.4. Contextual approximation $\leq_{CHF}$ *and* contextual equivalence $\sim_{CHF}$ *on processes are defined as $\leq_{CHF} := \leq_{\downarrow} \cap \leq_{\Downarrow}$ and $\sim_{CHF} := \leq_{CHF} \cap \geq_{CHF}$ where*

$$\begin{array}{lll} P_1 \leq_{\downarrow} P_2 & \textit{iff} & \forall \mathbb{D} \in \textit{PC} : \mathbb{D}[P_1]{\downarrow} \implies \mathbb{D}[P_2]{\downarrow} \\ P_1 \leq_{\Downarrow} P_2 & \textit{iff} & \forall \mathbb{D} \in \textit{PC} : \mathbb{D}[P_1]{\Downarrow} \implies \mathbb{D}[P_2]{\Downarrow} \end{array}$$

Transformations and Reduction Lengths in CHF We recall some results of [SSS11a] on the correctness of several program transformations for CHF. Moreover, for some specific cases we prove that the reduction length of a standard reduction is not increased by a transformation. These results will be necessary later when we show that the abstract machine is a correct evaluator for CHF.

A *program transformation* γ is a binary relation on processes. It is *correct* iff $\gamma \subseteq \sim_{CHF}$.

In Fig. 2 some program transformations are defined, where $\mathbb{C}$ is a process context with an expression hole. The general copying rule (gcp) allows to copy a binding into an arbitrary position, the transformation (cpx) is the special case where the copied expression is a variable, and the transformation (cpcxxL) is the special case of (gcp) where the copied

(gcp) $\mathbb{C}[x] \mid x = e \to \mathbb{C}[e] \mid x = e$

(cpx) $\mathbb{C}[x] \mid x = y \to \mathbb{C}[y] \mid x = y$, where y is a variable

(cpcxxL) $\widehat{\mathbb{L}}[x] \mid x = c\, y_1\ \ldots\ y_n \to \widehat{\mathbb{L}}[c\, y_1\ \ldots\ y_n] \mid x = c\, y_1\ \ldots\ y_n$,
where c is a constructor or a monadic operator, $\widehat{\mathbb{L}} \in \widehat{LC}$, and all y_i are variables

(gc) $\nu x_1, \ldots, x_n.(P \mid \mathtt{Comp}(x_1) \mid \ldots \mid \mathtt{Comp}(x_n)) \to P$
if for all $i \in \{1, \ldots, n\}$: $\mathtt{Comp}(x_i)$ is a binding $x_i = e_i$, an MVar $x_i\, \mathbf{m}\, e_i$,
or an empty MVar $x_i\, \mathbf{m}\, -$, and $x_i \notin FV(P)$.

Figure 2: The Transformations (gcp), (cpx), (cpcxxL), and (gc)

expression is a constructor application or a monadic operator where all arguments are variables and the target must be inside an $\widehat{LC}$-context. The rule (gc) performs garbage collection and thus allows to remove unused parts of the process.

Theorem 2.5 ([SSS11a])**.** *The reductions* (CHF, lunit), (CHF, nmvar), (CHF, fork), *and* (CHF, unIO) *are correct transformations. The transformations* (cp), (cpcx), (lbeta), (case), (seq), (mkbinds) *are correct as transformation in any context (i.e. the reduction rules in Fig. 1(c) where the context* $\mathbb{L}$ *is replaced by an arbitrary process context* $\mathbb{C}$ *with an expression hole) such that the scoping is not violated by the transformation. The transformations (gcp), (cpx), (cpcxxL), and (gc) are also correct.*

We introduce a special notion for reduction lengths:

Definition 2.6. *If* $P_0 \xrightarrow{CHF} P_1 \xrightarrow{CHF} \ldots \xrightarrow{CHF} P_n$ *where* P_n *is successful (*$P_n\Uparrow$*, resp.) and* $m \leq n$ *is the number of all reductions except for* (cp)*-reductions that copy a variable, then we write* $P_0{\downarrow^{[m,n]}}P_n$ *(*$P_0{\uparrow^{[m,n]}}P_n$*, resp.). We omit the process* P_n *if it is not of interest.*

In [Sab12] we show that the following properties on reduction lengths hold:

Proposition 2.7. *Let* P_1 *and* P_2 *be processes such that* $P_1 \xrightarrow{a} P_2$ *where* $a \in \{(CHF, cp), (gc), (cpx)\}$*. Then* $P_1{\downarrow^{[m,n]}} \implies P_2{\downarrow^{[m',n']}}$ *and* $P_1{\uparrow^{[m,n]}} \implies P_2{\uparrow^{[m',n']}}$ *where in both cases* $m' \leq m$ *and* $n' \leq n$*. If* $P_1 \xrightarrow{cpcxxL} P_2$*, then* $P_1{\downarrow^{[m,n]}} \implies P_2{\downarrow^{[m',n']}}$ *and* $P_1{\uparrow^{[m,n]}} \implies P_2{\uparrow^{[m',n']}}$ *where in both cases* $m' \leq m$*.*

3 Constructing an Abstract Machine for CHF

The goal of this section is to introduce an abstract machine for CHF. The construction of the machine is performed in three steps: first the machine $M1$ for evaluating pure functional expressions is introduced, then the machine is extended to handle monadic actions (called $IOM1$) and finally concurrency is added resulting in the machine $CIOM1$.

An Abstract Machine for Evaluating Pure Expressions The abstract machine $M1$ evaluates pure functional programs. It is analogous to Sestoft's machine mark 1 [Ses97]

but extended to operate also on `case`- and `seq`-expressions and to "functionally evaluate" monadic expressions, i.e. they are treated like ordinary constructor applications and *not* as actions. All of our abstract machines will only evaluate *simplified* expressions (analogous to *normalized expressions* in [Lau93, Ses97]):

Definition 3.1. *Simplified expressions Exp_S and simplified monadic expressions $MExp_S$ are built by the following grammar, where x, x_i are variables:*

$$
\begin{aligned}
e, e_i \in Exp_S \;&::= x \mid me \mid \lambda x.e \mid (e\ x) \mid c\ x_1 \ldots x_{\mathrm{ar}(c)} \mid \texttt{seq}\ e\ x \\
&\mid \texttt{letrec}\ x_1 = e_1\ \ldots\ x_n = e_n\ \texttt{in}\ e \\
&\mid \texttt{case}_T\ e\ \texttt{of}\ alt_{T,1} \ldots\ alt_{T,|T|} \ \textit{ where } alt_{T,i} = (c_{T,i}\ x_1 \ldots x_{\mathrm{ar}(c_{T,i})} \to e_i) \\
me \in MExp_S ::= \;&\texttt{return}\ x \mid x_1 \gg\!= x_2 \mid \texttt{future}\ x \\
&\mid \texttt{takeMVar}\ x \mid \texttt{newMVar}\ x \mid \texttt{putMVar}\ x_1\ x_2
\end{aligned}
$$

Simplified process $Proc_S$ are defined like processes Proc where all expressions are simplified expressions and additionally all MVars have only variables as content.

We first define the state of $M1$:

Definition 3.2. *A* state *of machine $M1$ is a tuple $(\mathcal{H}, e, \mathcal{S})$ where: $\mathcal{H}$ is a* heap*, i.e. a mapping of (finitely many) variables to expressions. To make the mapping explicit, we use the notation $\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$. We write $\mathcal{H}_1 \dot\cup \mathcal{H}_2$ for the disjoint union of the heaps $\mathcal{H}_1$ and $\mathcal{H}_2$. The second component, e, is a simplified expression. It is the currently evaluated expression. $\mathcal{S}$ is a stack, where allowed entries are $\#_{\mathsf{app}}(x)$, $\#_{\mathsf{seq}}(x)$, $\#_{\mathsf{case}}(alts)$, and $\#_{\mathsf{heap}}(x)$. We use list notation for stacks, i.e. $[]$ is the empty stack, and $a : \mathcal{S}$ is the stack with top entry a and tail $\mathcal{S}$.*

For a well-typed simplified expression e, the *initial state* of machine $M1$ is $(\emptyset, e, [])$. A state of $M1$ is a *final state* if it is of the form $(\mathcal{H}, v, [])$ where v is an abstraction, a constructor application, or a monadic expression. In Fig. 3(a) the *transition relation* $\xrightarrow{M1}$ of machine $M1$ is defined. The rules (pushApp), (pushSeq), and (pushAlts) perform unwinding to find the next redex. The corresponding contexts are stored on the stack. The rules (takeApp), (takeSeq), and (branch) perform beta-, seq-, and case-reduction. The rules (enter) and (update) are used to look up and restore (after a successful evaluation) bindings of the heap. The rule (mkBinds) moves local `letrec`-bindings into the (global) heap.

Compared to Sestoft's mark 1 we did some slight modifications (aside from handling `seq` and `case`): We did not include a rule (blackhole) for the case, that the redex is a variable which is not bound in the heap (e.g. this case may happen after trying to evaluate a recursive binding of the form $x \mapsto \texttt{seq}\ x\ x$). In our machine $M1$ there is simply no transition and the machine gets stuck. Another difference is in the (update) transition: While $M1$ allows to perform an update if the expression is a variable, Sestoft's mark 1 does not allow this transition. One reason for our modification is that later in the machine with IO-transitions ($IOM1$) we also must perform those updates, if the variables are names of MVars, e.g. for the process $y \Leftarrow \texttt{takeMVar}\ x \mid x = z \mid z\ \mathbf{m}\ v$ the name of the MVar z must be copied resulting in $y \Leftarrow \texttt{takeMVar}\ z \mid x = z \mid z\ \mathbf{m}\ v$. Finally, we do not explicitly perform α-renaming in our rules, but we assume that the distinct variable convention is always fulfilled and that necessary α-renamings are performed implicitly.

(pushApp) $(\mathcal{H}, (e\ x), \mathcal{S}) \xrightarrow{M1} (\mathcal{H}, e, \#_{\mathsf{app}}(x) : \mathcal{S})$

(pushSeq) $(\mathcal{H}, (\mathtt{seq}\ e\ x), \mathcal{S}) \xrightarrow{M1} (\mathcal{H}, e, \#_{\mathsf{seq}}(x) : \mathcal{S})$

(pushAlts) $(\mathcal{H}, (\mathtt{case}_T\ e\ \mathtt{of}\ alts), \mathcal{S}) \xrightarrow{M1} (\mathcal{H}, e, \#_{\mathsf{case}}(alts) : \mathcal{S})$

(takeApp) $(\mathcal{H}, \lambda x.e, \#_{\mathsf{app}}(y) : \mathcal{S}) \xrightarrow{M1} (\mathcal{H}, e[y/x], \mathcal{S})$

(takeSeq) $(\mathcal{H}, v, \#_{\mathsf{seq}}(y) : \mathcal{S}) \xrightarrow{M1} (\mathcal{H}, y, \mathcal{S})$, if v is an abstraction or a constructor app.

(branch) $(\mathcal{H}, (c\ x_1 \ldots x_n), \#_{\mathsf{case}}(\ldots (c\ y_1 \ldots y_n \to e) \ldots) : \mathcal{S}) \xrightarrow{M1} (\mathcal{H}, e[x_i/y_i]_{i=1}^n, \mathcal{S})$

(enter) $(\mathcal{H} \cup \{y \mapsto e\}, y, \mathcal{S}) \xrightarrow{M1} (\mathcal{H}, e, \#_{\mathsf{heap}}(y) : \mathcal{S})$

(update) $(\mathcal{H}, v, \#_{\mathsf{heap}}(y) : \mathcal{S}) \xrightarrow{M1} (\mathcal{H} \cup \{y \mapsto v\}, v, \mathcal{S})$
if v is an abstraction, a constructor app., a monadic operator, or a variable with $v \neq y$

(mkBinds) $(\mathcal{H}, \mathtt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \mathtt{in}\ e, \mathcal{S}) \xrightarrow{M1} (\mathcal{H} \cup \bigcup_{i=1}^n \{x_i \mapsto e_i\}, e, \mathcal{S})$

(a) Transition Relation $\xrightarrow{M1}$ of Machine $M1$

(M1) $(\mathcal{H}, \mathcal{M}, e, \mathcal{S}, \mathcal{I}) \xrightarrow{IOM1} (\mathcal{H}', \mathcal{M}', e', \mathcal{S}', \mathcal{I}')$
if $(\mathcal{H}, e, \mathcal{S}) \xrightarrow{M1} (\mathcal{H}', e', \mathcal{S}')$ on machine $M1$

(newMVar) $(\mathcal{H}, \mathcal{M}, \mathtt{newMVar}\ x, [], \mathcal{I}) \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M} \cup \{y\ \mathbf{m}\ x\}, \mathtt{return}\ y, [], \mathcal{I})$
where y is a fresh variable

(takeMVar) $(\mathcal{H}, \mathcal{M} \cup \{x\ \mathbf{m}\ y\}, x, [], \#_{\mathsf{take}} : \mathcal{I}) \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M} \cup \{x\ \mathbf{m}\ -\}, \mathtt{return}\ y, [], \mathcal{I})$

(putMVar) $(\mathcal{H}, \mathcal{M} \cup \{x\ \mathbf{m}\ -\}, x, [], \#_{\mathsf{put}}(y) : \mathcal{I}) \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M} \cup \{x\ \mathbf{m}\ y\}, \mathtt{return}\ (), [], \mathcal{I})$

(pushTake) $(\mathcal{H}, \mathcal{M}, \mathtt{takeMVar}\ x, [], \mathcal{I}) \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M}, x, [], \#_{\mathsf{take}} : \mathcal{I})$

(pushPut) $(\mathcal{H}, \mathcal{M}, \mathtt{putMVar}\ x\ y, [], \mathcal{I}) \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M}, x, [], \#_{\mathsf{put}}(y) : \mathcal{I})$

(pushBind) $(\mathcal{H}, \mathcal{M}, x \gg\!= y, [], \mathcal{I}) \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M}, x, [], \#_{\gg\!=}(y) : \mathcal{I})$

(lunit) $(\mathcal{H}, \mathcal{M}, \mathtt{return}\ x, [], \#_{\gg\!=}(y) : \mathcal{I}) \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M}, (y\ x), [], \mathcal{I})$

(b) Transition Relation $\xrightarrow{IOM1}$ of Machine $IOM1$

(unIO) $(\mathcal{H}, \mathcal{M}, \mathcal{T} \cup \{(x, (\mathtt{return}\ y), [], [])\}) \xrightarrow{CIOM1} (\mathcal{H} \cup \{x \mapsto y\}, \mathcal{M}, \mathcal{T})$
if thread named x is not the main-thread

(fork) $(\mathcal{H}, \mathcal{M}, \mathcal{T} \cup \{(x, (\mathtt{future}\ y), [], \mathcal{I})\})$
$\xrightarrow{CIOM1} (\mathcal{H}, \mathcal{M}, \mathcal{T} \cup \{(x, (\mathtt{return}\ z), [], \mathcal{I}), (z, y, [], [])\})$
where z is a fresh variable

(IOM1) $(\mathcal{H}, \mathcal{M}, \mathcal{T} \cup \{(x, e, \mathcal{S}, \mathcal{I})\}) \xrightarrow{CIOM1} (\mathcal{H}', \mathcal{M}', \mathcal{T} \cup \{(x, e', \mathcal{S}', \mathcal{I}')\})$
if $(\mathcal{H}, \mathcal{M}, e, \mathcal{S}, \mathcal{I}) \xrightarrow{IOM1} (\mathcal{H}', \mathcal{M}', e', \mathcal{S}', \mathcal{I}')$ on machine $IOM1$.
The rule is only used if (fork) or (unIO) is not applicable for the thread named x.

(c) Transition Relation $\xrightarrow{CIOM1}$ of Machine $CIOM1$

Figure 3: Transition Relations of the Machines $M1$, $IOM1$, and $CIOM1$

Example 3.3. *We demonstrate the evaluation of machine $M1$:*

$$
\begin{array}{ll}
 & (\emptyset, \texttt{letrec}\ x_1 = (\lambda y.y)\ w,\ x_2 = \texttt{takeMVar}\ x_1, x_3 = x_2\ \texttt{in}\ (\lambda z.z)\ x_3, [])\\
\xrightarrow{M1,\text{mkBinds}} & (\{x_1 \mapsto (\lambda y.y)\ w, x_2 \mapsto \texttt{takeMVar}\ x_1, x_3 \mapsto x_2\}, (\lambda z.z)\ x_3, [])\\
\xrightarrow{M1,\text{pushApp}} & (\{x_1 \mapsto (\lambda y.y)\ w, x_2 \mapsto \texttt{takeMVar}\ x_1, x_3 \mapsto x_2\}, \lambda z.z, [\#_{\mathsf{app}}(x_3)])\\
\xrightarrow{M1,\text{takeApp}} & (\{x_1 \mapsto (\lambda y.y)\ w, x_2 \mapsto \texttt{takeMVar}\ x_1, x_3 \mapsto x_2\}, x_3, [])\\
\xrightarrow{M1,\text{enter}} & (\{x_1 \mapsto (\lambda y.y)\ w, x_2 \mapsto \texttt{takeMVar}\ x_1\}, x_2, [\#_{\mathsf{heap}}(x_3)])\\
\xrightarrow{M1,\text{update}} & (\{x_1 \mapsto (\lambda y.y)\ w, x_2 \mapsto \texttt{takeMVar}\ x_1, x_3 \mapsto x_2\}, x_2, [])\\
\xrightarrow{M1,\text{enter}} & (\{x_1 \mapsto (\lambda y.y)\ w, x_3 \mapsto x_2\}, \texttt{takeMVar}\ x_1, [\#_{\mathsf{heap}}(x_2)])\\
\xrightarrow{M1,\text{update}} & (\{x_1 \mapsto (\lambda y.y)\ w, x_3 \mapsto x_2, x_2 \mapsto \texttt{takeMVar}\ x_1\}, \texttt{takeMVar}\ x_1, [])
\end{array}
$$

The last state is a final state, since $M1$ treats monadic operators like values.

Extending $M1$ by Monadic I/O We will extend the machine $M1$, such that MVars and operations on MVars can be performed. We have to implement the operations of the monad, i.e. `return`, $>\!\!>\!\!=$ and the operations `takeMVar`, `putMVar`, `newMVar` to access and create MVars. The state of the machine is extended by two components: a set of MVars which models the memory and a further stack – called IO-stack – which allows a clean separation between monadic and functional evaluation. An *IO-stack* is a stack where the following entries are allowed: The symbol $\#_{\mathsf{take}}$ to store a `takeMVar` operation, entries of the form $\#_{\mathsf{put}}(x)$ to store a `putMVar`-operation, where x is the new (to-be-written) content of the MVar, and $\#_{>\!\!>\!\!=}(y)$ to store a $>\!\!>\!\!=$-operation, where y is the right argument of $>\!\!>\!\!=$.

Definition 3.4. *A state of the machine $IOM1$ is a tuple $(\mathcal{H}, \mathcal{M}, e, \mathcal{S}, \mathcal{I})$ where heap $\mathcal{H}$, expression e, and stack $\mathcal{S}$ are as before (in machine $M1$). $\mathcal{M}$ is a set of MVars with variables as content: a filled MVar is written as $x\ \mathbf{m}\ y$, and an empty MVar is written as $x\ \mathbf{m}\ -$. $\mathcal{I}$ is an IO-stack.*

We only consider the evaluation of expressions of `IO`-type. For computing an expression $e :: \texttt{IO}\ \tau$ the machine $IOM1$ starts with state $(\emptyset, \emptyset, e, [], [])$. A state is a *final state* if both stacks are empty and the evaluated expression is of the form $(\texttt{return}\ x)$.

The transition relation $\xrightarrow{IOM1}$ of the machine $IOM1$ is defined in Figure 3(b). The first rule lifts all transitions of $M1$ to machine $IOM1$. The remaining rules have in common, that they require the (usual) stack $\mathcal{S}$ to be empty. That is how functional evaluation is separated from monadic computation: as long as the usual stack is filled, functional evaluation is performed and if the usual stack is empty, then monadic computations are performed. The rule (newMVar) creates a new MVar and returns its name. The rule (takeMVar) takes the content of a filled MVar. There is no rule for the case that the MVar is already empty. In this case the machine gets stuck. Performing the take-operation requires that the to-be-evaluated expression is already the name of the MVar. That is why first (pushTake) pushes the take-operation on the IO-stack and thus forces the argument to be evaluated first. The rules (putMVar) and (pushPut) are the corresponding rules for performing a `putMVar`-operation: First (pushPut) enforces the first argument to be evaluated (to get the name of the MVar),

then either (putMVar) is performed to fill the MVar (if it is empty) or the machine gets stuck, if the MVar is already filled. For implementing the monadic sequencing operator $\gg\!=$, the action on the left hand side is performed first. Hence the (pushBind)-operation stores the second argument on the IO-stack. When the execution of the first action ends successfully with $(\texttt{return}\ x)$, then rule (lunit) evaluates the $\gg\!=$-operator.

A single thread $y \Leftarrow \mathbb{M}[\mathbb{F}[e]]$ of *CHF* corresponds to a machine state of *IOM1* as follows: the IO-stack holds the corresponding $\mathbb{M}$-context of the expression and also the `takeMVar`- or `putMVar`-operation on the top-level of the $\mathbb{F}$-context. The call-by-name evaluation context $\mathbb{E}$ inside the $\mathbb{F}$-context is stored on the usual stack.

Since we only evaluate well-typed expressions, the following lemma holds:

Lemma 3.5. *For any machine state of IOM1 which is reachable from a start state for a well-typed expression $e :: \texttt{IO}\ \tau$, the IO-stack is of the following form: All entries are of the form $\#_{\gg}(x)$ except for the top-element which also may be $\#_{\mathsf{take}}$ or $\#_{\mathsf{put}}(x)$.*

Example 3.6. *We again consider the expression of Example 3.3 and its execution on machine IOM1, where we assume that the set $\mathcal{M}$ contains a filled MVar $w\,\mathbf{m}\,c$. The first eight transitions are as on machine M1, where the MVars and the IO-stack are irrelevant:*

$$(\emptyset, \{w\,\mathbf{m}\,c\}, \texttt{letrec}\ x_1 = (\lambda y.y)\ w,\ x_2 = \texttt{takeMVar}\ x_1, x_3 = x_2\ \texttt{in}\ (\lambda z.z)\ x_3, [], [])$$
$$\xrightarrow{\textit{IOM1},*} (\{x_1 \mapsto (\lambda y.y)\ w, x_3 \mapsto x_2, x_2 \mapsto \texttt{takeMVar}\ x_1\}, \{w\,\mathbf{m}\,c\}, \texttt{takeMVar}\ x_1, [], [])$$

Now machine IOM1 proceeds as follows:

$$\xrightarrow{\textit{IOM1},\text{pushTake}} (\{x_1 \mapsto (\lambda y.y)\ w, x_3 \mapsto x_2, x_2 \mapsto \texttt{takeMVar}\ x_1\}, \{w\,\mathbf{m}\,c\}, x_1, [], [\#_{\mathsf{take}}])$$
$$\xrightarrow{\textit{IOM1},\text{enter}} (\{x_3 \mapsto x_2, x_2 \mapsto \texttt{takeMVar}\ x_1\}, \{w\,\mathbf{m}\,c\}, (\lambda y.y)\ w, [\#_{\mathsf{heap}}(x_1)], [\#_{\mathsf{take}}])$$
$$\xrightarrow{\textit{IOM1},\text{pushApp}} (\{x_3 \mapsto x_2, x_2 \mapsto \texttt{takeMVar}\ x_1\}, \{w\,\mathbf{m}\,c\}, \lambda y.y, [\#_{\mathsf{app}}(w), \#_{\mathsf{heap}}(x_1)], [\#_{\mathsf{take}}])$$
$$\xrightarrow{\textit{IOM1},\text{takeApp}} (\{x_3 \mapsto x_2, x_2 \mapsto \texttt{takeMVar}\ x_1\}, \{w\,\mathbf{m}\,c\}, w, [\#_{\mathsf{heap}}(x_1)], [\#_{\mathsf{take}}])$$
$$\xrightarrow{\textit{IOM1},\text{update}} (\{x_3 \mapsto x_2, x_2 \mapsto \texttt{takeMVar}\ x_1, x_1 \mapsto w\}, \{w\,\mathbf{m}\,c\}, w, [], [\#_{\mathsf{take}}])$$
$$\xrightarrow{\textit{IOM1},\text{takeMVar}} (\{x_3 \mapsto x_2, x_2 \mapsto \texttt{takeMVar}\ x_1, x_1 \mapsto w\}, \{w\,\mathbf{m}\,-\}, \texttt{return}\ c, [], [])$$

Adding Concurrency Constructing the concurrent machine *CIOM1* from the sequential machine *IOM1* is easy, since most of the parts of the machine *IOM1* can be reused. Instead of evaluating a single expression, the machine *CIOM1* will evaluate several expressions in several threads. Any such thread consists of a to-be-evaluated expression, a stack, and an IO-stack. Moreover, since threads represent futures, every thread has a name (a variable). There is one unique distinguished thread, the *main thread*. If the main-thread is successfully evaluated, then the whole machine stops. Further components of the machine *CIOM1* are the heap $\mathcal{H}$ and the set of MVars $\mathcal{M}$ which are globally shared over all threads. For the transition relation of the machine *CIOM1* a single thread is non-deterministically selected and the (thread-local) transition is performed for the selected thread. For this thread-local transition we can reuse the transition relation of the machine *IOM1*. There are two exceptions: If the monadic operation `future` spawns a new thread, and if a thread finishes its evaluation such that its result can be shared in the heap.

Definition 3.7. *A* thread *(or* future*, alternatively) of the machine $CIOM1$ is a 4-tuple $(x, e, \mathcal{S}, \mathcal{I})$ where x is a variable, called the* name *of the future, e is a simplified expression which is evaluated by the thread, $\mathcal{S}$ is a stack, and $\mathcal{I}$ is an IO-stack. A future can be distinguished as a main-thread, which we sometimes write as $(x, e, \mathcal{S}, \mathcal{I})^{main}$.*

A state *of machine $CIOM1$ is a 3-tuple $(\mathcal{H}, \mathcal{M}, \mathcal{T})$ where $\mathcal{H}$ is a heap of shared bindings, $\mathcal{M}$ is a set of MVars, and $\mathcal{T}$ is a set of threads.*

Definition 3.8. *For a simplified expression $e :: \texttt{IO}\ \tau$ the* start state $\mathsf{Init}(e)$ *of machine $CIOM1$ is $(\emptyset, \emptyset, \mathcal{T})$ where $\mathcal{T} = \{(x, e, [], [])^{main}\}$ and x is a fresh variable ($x \notin FV(e)$).*

A state of the machine $CIOM1$ is a final state *if the main-thread is of the form $(y, \texttt{return}\ x, [], [])^{main}$ where y and x may be equal.*

Definition 3.9. *The transition relation $\xrightarrow{CIOM1}$ of machine $CIOM1$ is shown in Fig. 3(c). For one step a thread is selected which may proceed. This selection is performed nondeterministically over all threads. Note that threads which cannot proceed are not selected. Those threads are about to evaluate a variable which is not bound in the heap, or try to perform a* (takeMVar)- *or* (putMVar)*-transition on an empty or filled MVar. We also assume that transitions are not applicable to final states.*

When a thread successfully finishes its computation, the rule (unIO) removes the thread and stores the result in the heap by a new binding. Note that other threads which want to access the value of a future x will not be selected for transition until the result becomes available as a binding in the heap. The rule (fork) evaluates a `future`-operation and spawns a new thread. In all other cases the rule (IOM1) is used which lifts the transition relation $\xrightarrow{IOM1}$ of $IOM1$ to the concurrent machine $CIOM1$.

Note that for a real implementation one would require some kind of fairness and thus for instance organize the set of threads as a priority-queue of threads.

Definition 3.10. *A state S is* valid*, if there exists $e :: \texttt{IO}\ \tau$ such that $\mathsf{Init}(e) \xrightarrow{CIOM1,*} S$.*

We only consider valid states in the following. It is easy to verify that for any valid state of $CIOM1$ all introduced variables (names of MVars, left hand sides of heap bindings, and names of threads) are pairwise distinct, all $\#_{\mathsf{heap}}(x)$-entries in stacks are pairwise distinct, and all the variables x in such entries do not occur as a left hand side in the heap.

4 Correctness of the Abstract Machine

In this section we will show that the abstract machine $CIOM1$ is a correct evaluator for CHF, that is for all expressions $e :: \texttt{IO}\ \tau$ may- and should-convergence of CHF coincide with may- and should-convergence of the machine $CIOM1$ where e is simplified before the evaluation. Indeed we will not only consider expressions and will work with processes in most of our proofs. As a simplification we assume that in CHF for the evaluation of a process all ν-binders are dropped and that reduction does not introduce ν-binders. Instead corresponding α-renamings are performed implicitly to represent the according scopes.

We first show that it is correct to take into account simplified expressions and processes, only. The first translation shares all necessary parts to derive simplified processes, i.e. general processes can be transformed into simplified processes by creating new bindings.

Definition 4.1. *The function $\sigma :: Proc \to Proc_S$ translates processes into simplified processes. It is defined to be homomorphic over the term structure (e.g. $\sigma(P_1 \mid P_2) := \sigma(P_1) \mid \sigma(P_2)$, etc.) except for the following cases:*

$$
\begin{array}{lcl}
\sigma(e_1\ e_2) & := & \texttt{letrec}\ x = \sigma(e_2)\ \texttt{in}\ (\sigma(e_1)\ x) \\
\sigma(c\ e_1\ \ldots\ e_n) & := & \texttt{letrec}\ x_1 = \sigma(e_1), \ldots, x_n = \sigma(e_n)\ \texttt{in}\ c\ x_1\ \ldots\ x_n \\
 & & \textit{if } c \textit{ is a constructor, or a monadic operator} \\
\sigma(\texttt{seq}\ e_1\ e_2) & := & \texttt{letrec}\ x = \sigma(e_2)\ \texttt{in}\ \texttt{seq}\ \sigma(e_1)\ x \\
\sigma(x\ \mathbf{m}\ e) & := & x\ \mathbf{m}\ y \mid y = \sigma(e)
\end{array}
$$

The results in [SSS11a] imply that the translation σ preserves contextual equivalence:

Theorem 4.2. *For all processes $P \in Proc$: $P \sim_{CHF} \sigma(P)$.*

We define may- and should-convergence based on the machine transition of $CIOM1$:

Definition 4.3. *A valid state S* may-converges *($S{\downarrow_{CIOM1}}$) iff there exists a final state S' such that $S \xrightarrow{CIOM1,*} S'$; and S* should-converges *($S{\Downarrow_{CIOM1}}$) iff $\forall S' : S \xrightarrow{CIOM1,*} S' \implies S'{\downarrow_{CIOM1}}$. An expression $e :: \texttt{IO}\ \tau$* may-converges on *CIOM1 ($e{\downarrow_{CIOM1}}$) iff $\mathsf{Init}(\sigma(e)){\downarrow_{CIOM1}}$, and e* should-converges *on CIOM1 ($e{\Downarrow_{CIOM1}}$) iff $\mathsf{Init}(\sigma(e)){\Downarrow_{CIOM1}}$. We write $e{\Uparrow_{CIOM1}}$ iff $\neg(e{\downarrow_{CIOM1}})$ and $e{\uparrow_{CIOM1}}$ iff $\neg(e{\Downarrow_{CIOM1}})$.*

Note that if we would restrict evaluation to *fair* evaluations only, i.e. forbidding (infinite) reductions sequences where an executable thread is ignored infinitely long, then the induced predicates of may- and should-convergence are unchanged (see also e.g. [Sab08, SSS11a]). Thus for reasoning it is not necessary to explicitly treat fairness.

We will now define the translation ρ which translates valid machine states of $CIOM1$ into processes. Note that the resulting process is not necessarily simplified. In abuse of notation we allow also non-simplified expressions inside the machine state during the translation.

Definition 4.4. *Let $(\mathcal{H}, \mathcal{M}, \mathcal{T}) = (\bigcup_{i=1}^{n}\{x_i \mapsto e_1\}, \{m_1, \ldots, m_{n'}\}, \{T_1, \ldots, T_{n''}\})$ be a valid machine state of CIOM1 where m_i are MVars and T_i are threads.*
Then $\rho(\mathcal{H}, \mathcal{M}, \mathcal{T}) := x_1 = e_1 \mid \ldots \mid x_n = e_n \mid m_1 \mid \ldots \mid m_{n'} \mid \rho(T_1) \mid \ldots \mid \rho(T_{n''})$ where a single thread T_i is translated as follows:

$$
\begin{array}{ll}
\rho(y, e, \#_{\mathsf{app}}(x) : \mathcal{S}, \mathcal{I}) & := \rho(y, e\ x, \mathcal{S}, \mathcal{I}) \\
\rho(y, e, \#_{\mathsf{seq}}(x) : \mathcal{S}, \mathcal{I}) & := \rho(y, \texttt{seq}\ e\ x, \mathcal{S}, \mathcal{I}) \\
\rho(y, e, \#_{\mathsf{heap}}(x) : \mathcal{S}, \mathcal{I}) & := x = e \mid \rho(y, x, \mathcal{S}, \mathcal{I}) \\
\rho(y, e, \#_{\mathsf{case}}(alts) : \mathcal{S}, \mathcal{I}) & := \rho(y, \texttt{case}\ e\ \texttt{of}\ alts, \mathcal{S}, \mathcal{I}) \\
\rho(y, e, [], \#_{\gg=}(x) : \mathcal{I}) & := \rho(y, e \gg= x, [], \mathcal{I}) \\
\rho(y, e, [], \#_{\mathsf{take}} : \mathcal{I}) & := \rho(y, \texttt{takeMVar}\ e, [], \mathcal{I}) \\
\rho(y, e, [], \#_{\mathsf{put}}(x) : \mathcal{I}) & := \rho(y, \texttt{putMVar}\ e\ x, [], \mathcal{I}) \\
\rho(y, e, [], []) & := y \overset{\mathsf{main}}{\Longleftarrow} e, \textit{ if } y \textit{ is a main-thread, and } y \Leftarrow e, \textit{ otherwise}
\end{array}
$$

Lemma 4.5. *Let S be a valid machine state with $S \xrightarrow{CIOM1} S'$. Then either $\rho(S) = \rho(S')$ or $\rho(S) \xrightarrow{CHF} \xrightarrow{cpx,*} \xrightarrow{gc,*} \rho(S')$.*

Proof. This follows by inspecting all cases (see [Sab12]). The (cpx) and (gc) transformations are necessary to remove variable-to-variable bindings which are introduced in CHF by (lbeta), (case), and (cpcx) but not by the corresponding transitions (takeApp), (branch), and (update). □

Proposition 4.6. *For every valid state S of CIOM1: $S{\downarrow_{CIOM1}} \implies \rho(S){\downarrow}$.*

Proof. Let $S_n{\downarrow_{CIOM1}}$, i.e. $S_n \xrightarrow{CIOM1} \ldots \xrightarrow{CIOM1} S_0$ where S_0 is a final state. We use induction on n: If $n = 0$, then S_n is a final state and $\rho(S_n)$ is successful. For the induction step assume that $\rho(S_{n-1}){\downarrow}$. The analysis in Lemma 4.5 shows that either $\rho(S_n) = \rho(S_{n-1})$, $\rho(S_n) \xrightarrow{CHF} \rho(S_{n-1})$, or $\rho(S_n) \xrightarrow{CHF} P \sim_{CHF} \rho(S_{n-1})$ (since (cpx) and (gc) are correct program transformations, see Theorem 2.5). For the first two cases obviously $\rho(S_n){\downarrow}$, for the third case $S_{n-1}{\downarrow}$ and contextual equivalence imply that $P{\downarrow}$ and thus also $\rho(S_n){\downarrow}$. □

Given a state S and a reduction of the corresponding process, say $\rho(S) \xrightarrow{CHF} P$, we now try to find a sequence of corresponding machine transitions for S.

Lemma 4.7. *Let S be a valid machine state, and let $\rho(S) \xrightarrow{CHF} P$. Then there exists a valid state S' with $S \xrightarrow{CIOM1,*} S'$ such that one of the following properties holds: (1) $\rho(S') = P$; or (2) $P \xrightarrow{CHF,cp} \rho(S')$; or (3) in case of a $(CHF, cpcx)$-reduction $P \xrightarrow{cpcxxL} \xrightarrow{cpx,*} \xrightarrow{gc,*} \rho(S')$; or (4) $P \xrightarrow{cpx,*} \xrightarrow{gc,*} \rho(S')$.*

Proof. We give a brief description, details are in [Sab12]. Several transitions are necessary to find the corresponding redex using the transitions (pushBind), (pushApp), (pushSeq), (pushAlts), and (enter). For the first case a machine transition corresponds to standard reduction in CHF. The second and third case may occur if a (cp) or (cpcx) reduction is performed: then perhaps the corresponding heap binding in the machine is under evaluation of the wrong thread and the machine must perform two (update) transitions, where one corresponds to the (cp) (or (cpcx)) reduction, and the other one is also a (cp) standard reduction or a (cpcxxL)-transformation. If a constructor was shared by a $(CHF, cpcx)$-reduction, then the generated variable-to-variable bindings must be inlined and removed by performing a sequence of (cpx) and (gc) transformations. Case (4) describes a necessary removal of variable-to-variable bindings which are introduced by a (lbeta)- or (case)-reduction. □

Proposition 4.8. *For every valid machine state S of CIOM1: $\rho(S){\downarrow} \implies S{\downarrow_{CIOM1}}$.*

Proof. Let $P_n{\downarrow^{[m,n]}}P_0$, i.e. $P_n \xrightarrow{CHF} P_{n-1} \xrightarrow{CHF} \ldots \xrightarrow{CHF} P_0$ where P_0 is successful, and m is the number of all reductions except of (cp)-reductions that copy a variable. We use induction on the pair (m, n), ordered lexicographically. For $n = 0$ the claim holds, since only final machine states are translated into successful processes. For the induction step assume that the claim holds for all $(m', n') < (m, n)$. We apply Lemma 4.7 to the reduction $\rho(S_n) = P_n \xrightarrow{CHF} P_{n-1}$ where $P_{n-1}{\downarrow^{[m',n-1]}}$ such that either $m' = m$ (if the

reduction is also a (cpx)-transformation), or $m' = m - 1$ (in all other cases). This shows $S_n \xrightarrow{CIOM1,*} S'$ by the following cases: (i) $\rho(S') = P_{n-1}$: Then $S_n{\downarrow_{CIOM1}}$ by the induction hypothesis. (ii) $P_{n-1} \xrightarrow{CHF,cp} \rho(S')$, or $P_{n-1} \xrightarrow{cpx,*} \xrightarrow{gc,*} \rho(S')$. Then Proposition 2.7 shows that $\rho(S'){\downarrow^{[m'',n'']}}$ where $(m'', n'') < (m, n)$. Applying the induction hypothesis to $\rho(S')$ yields $S'{\downarrow_{CIOM1}}$ and thus also $S_n{\downarrow_{CIOM1}}$. (iii) $P_{n-1} \xrightarrow{cpcxxL} \xrightarrow{cpx,*} \xrightarrow{gc,*} \rho(S')$. Then the equation $m' = m - 1$ must hold, since the standard reduction is $(CHF, cpcx)$. Proposition 2.7 shows that $\rho(S'){\downarrow^{[m'',n'']}}$ where $(m'', n'') < (m, n)$ and thus we can apply the induction hypothesis to $\rho(S')$ and have $S'{\downarrow_{CIOM1}}$ and thus also $S_n{\downarrow_{CIOM1}}$. □

Since $\neg{\downarrow} = {\Uparrow}$ and $\neg{\downarrow_{CIOM1}} = {\Uparrow_{CIOM1}}$, Propositions 4.6 and 4.8 also imply:

Lemma 4.9. *For every valid machine state S of CIOM1: $\rho(S){\Uparrow} \iff S{\Uparrow_{CIOM1}}$.*

Proposition 4.10. *For every valid machine state S of CIOM1: $\rho(S){\Downarrow} \iff S{\Downarrow_{CIOM1}}$.*

Proof. The claim is equivalent to $\rho(S){\uparrow} \iff S{\uparrow_{CIOM1}}$. Both directions can be proved by induction analogously to the proofs for may-convergence in Propositions 4.6 and 4.8 except for the base cases of the inductions which are covered by Lemma 4.9. □

Theorem 4.11. *For every expression $e :: \texttt{IO}\ \tau$ the equivalences $e{\downarrow} \iff e{\downarrow_{CIOM1}}$ and $e{\Downarrow} \iff e{\Downarrow_{CIOM1}}$ hold.*

Proof. This follows from Propositions 4.6,4.8, and 4.10 and since for any well-typed expression $e :: \texttt{IO}\ \tau$ we have $\rho(\mathsf{Init}(\sigma(e))) = x \overset{\mathsf{main}}{\Longleftarrow} \sigma(e) \sim_{CHF} x \overset{\mathsf{main}}{\Longleftarrow} e$ where the last equivalence holds by Theorem 4.2. □

5 Conclusion

We introduced the concurrent abstract machine *CIOM1* for evaluation of *CHF*-programs and showed that the machine is a correct evaluator w.r.t. the semantics of the process calculus *CHF*. Further work is to optimize the machine, e.g. by following the modifications presented in [Ses97] (avoiding substitutions by using closures, using a nameless representation by de Bruijn-indices, etc.) and showing correctness of them. Another direction is to analyze how to map the threads of *CIOM1* to a multicore architecture.

Acknowledgments I thank Manfred Schmidt-Schauß for reading this paper and for discussions on this paper. I also thank the anonymous reviewers for their valuable comments.

References

[AHH+05] E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for declarative multi-paradigm languages. *J. Symb. Comput.*, 40:795–829, 2005.

[BFKT00] C. A. Baker-Finch, D. J. King, and P. W. Trinder. An operational semantics for parallel lazy evaluation. In *5th ICFP*, pp. 162–173. ACM, 2000.

[CHS05] A. Carayol, D. Hirschkoff, and D. Sangiorgi. On the representation of McCarthy's amb in the Pi-calculus. *Theoret. Comput. Sci.*, 330(3):439–473, 2005.

[DF07] R. Douence and P. Fradet. The next 700 Krivine machines. *Higher Order Symbol. Comput.*, 20:237–255, 2007.

[Lau93] J. Launchbury. A natural semantics for lazy evaluation. In *20th POPL*, pp. 144–154. ACM, 1993.

[Mil99] R. Milner. *Communicating and mobile systems: the π-calculus*. CUP, 1999.

[Mor98] A. Moran. *Call-by-name, call-by-need, and McCarthy's Amb*. PhD thesis, Dept. of Comp. Science, Chalmers university, Sweden, 1998.

[MSC99] A. Moran, D. Sands, and M. Carlsson. Erratic Fudgets: A semantic theory for an embedded coordination language. In *Coordination '99*, *LNCS* 1594, pp. 85–102. 1999.

[NSSSS07] J. Niehren, D. Sabel, M. Schmidt-Schauß, and J. Schwinghammer. Observational semantics for a concurrent lambda calculus with reference cells and futures. *Electron. Notes Theor. Comput. Sci.*, 173:313–337, 2007.

[Pey01] S. Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction*, pp. 47–96. IOS-Press, 2001.

[PGF96] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *23th POPL*, pp. 295–308. ACM, 1996.

[PS09] S. Peyton Jones and S. Singh. A tutorial on parallel and concurrent programming in Haskell. In *6th AFP*, pp. 267–305. Springer, 2009.

[RV07] A. Rensink and W. Vogler. Fair testing. *Inform. and Comput.*, 205(2):125–198, 2007.

[Sab08] D. Sabel. *Semantics of a call-by-need lambda calculus with McCarthy's amb for program equivalence*. Dissertation, Goethe-Universität Frankfurt Germany, 2008.

[Sab12] D. Sabel. An abstract machine for Concurrent Haskell with futures. Frank report 48, Institut für Informatik, Goethe-Universität Frankfurt am Main, 2012. http://www.ki.informatik.uni-frankfurt.de/papers/frank/.

[Ses97] P. Sestoft. Deriving a lazy abstract machine. *J. Funct. Progr.*, 7(3):231–264, 1997.

[SSS08] D. Sabel and M. Schmidt-Schauß. A call-by-need lambda-calculus with locally bottom-avoiding choice: context lemma and correctness of transformations. *Math. Structures Comput. Sci.*, 18(03):501–553, 2008.

[SSS10] M. Schmidt-Schauß and D. Sabel. Closures of may-, should- and must-convergences for contextual equivalence. *Inform. Process. Lett.*, 110(6):232 – 235, 2010.

[SSS11a] D. Sabel and M. Schmidt-Schauß. A contextual semantics for Concurrent Haskell with futures. In *13th PPDP*, pp. 101–112, ACM, 2011.

[SSS11b] D. Sabel and M. Schmidt-Schauß. On conservativity of Concurrent Haskell. Frank report 47, Institut für Informatik, Goethe-Universität Frankfurt am Main, 2011. http://www.ki.informatik.uni-frankfurt.de/papers/frank/.

[SW01] D. Sangiorgi and D. Walker. *The π-calculus: a theory of mobile processes*. CUP, 2001.