Embedding the Pi-Calculus into a Concurrent Functional Programming Language *

Manfred Schmidt-Schauss[1] and David Sabel[1,2]

[1] Goethe-University, Frankfurt, Germany
[2] LMU Munich, Germany

Technical Report Frank-60

Research Group for Artificial Intelligence and Software Technology
Institut für Informatik,
Fachbereich Informatik und Mathematik,
Johann Wolfgang Goethe-Universität,
D-60325 Frankfurt, Germany

February 12, 2019

Abstract. The synchronous pi-calculus is translated into a core language of Concurrent Haskell extended by futures (CHF). The translation simulates the synchronous message-passing of the pi-calculus by sending messages and adding synchronization using Concurrent Haskell's mutable shared-memory locations (MVars). The semantic criterion is a contextual semantics of the pi-calculus and of CHF using may- and should-convergence as observations. The results are equivalence with respect to the observations, full abstraction of the translation of closed processes, and adequacy of the translation on open pocesses. The translation transports the semantics of the pi-calculus processes under rather strong criteria, since error-free programs are translated into error-free ones, and programs without non-deterministic error possibilities are also translated into programs without non-deterministic error-possibilities. This investigation shows that CHF embraces the expressive power and the concurrency capabilities of the pi-calculus.

Keywords: pi-calculus, functional programming; concurrency, adequate translations

1 Introduction

We are interested in concurrent and at the same time declarative program calculi and their expressiveness as models for concurrent programming languages.

The well-known π-calculus [MPW92,Mil99,SW01b] is a minimal model for *mobile and concurrent processes*. Data-flow is possible by communication between processes, i.e. by passing messages between them. Channel names are sent as messages, links between processes can be dynamically created and removed, and also new processes may be created, which together makes processes mobile. The interest in the π-calculus is not only due to the fact that it is used and extended for a lot of applications. Examples are the Spi-calculus [AG97] for reasoning about cryptographic protocols, the stochastic π-calculus [Pri95] with applications in molecular biology, and the join calculus [Lan96,FG02] for distributed computing. The π-calculus also permits the study of intrinsic principles and semantics of concurrency, of concurrent programming and the inherent nondeterministic behavior of mobile and communicating processes.

The investigated variant of the π-calculus in this paper is the synchronous π-calculus with replication, but without sums, matching operators, or recursion.

* The first and third authors are supported by the Deutsche Forschungsgemeinschaft (DFG) under grant SCHM 986/11-1, and the second author is supported by the Deutsche Forschungsgemeinschaft (DFG) under grant SA2908/3-1.

Almost every programming language nowadays permits forms of parallelism, implicit, or explicit parallelism like threads, or concurrency constructs, and process management. An issue is whether the expressiveness of the π-calculus is also available in sufficiently interesting higher-order and concurrent programming languages. An example for such a calculus and also a programming language is Concurrent Haskell [PGF96,com19]. For our study, we will use the language *CHF* [SSS11,SSS12] that is an extended variant of Concurrent Haskell and which permits also futures in order to increase the declarativeness of such languages, and where studies on the observational (contextual) semantics are available.

The contextual semantics of concurrent languages is a generalization of a black-box testing principle of programs: the semantic equality of two (programmed) functions f, g holds, if for all inputs n, $f(n)$ and $g(n)$ compute the same output. An equivalent observation is whether a program successfully terminates (converges) or not. For processes, which are nondeterministic programs, the patterns of behavior are usually more complex. A generalization of the black-box testing is to compare the processes in process-contexts, and to use two observations: *may-convergence* – at least one execution path terminates successfully, and *should-convergence* – every intermediate state may-converges. An alternative nondeterministic observation is must-convergence (all execution paths terminate successfully). The advantage of equivalence notions based on should-convergence are their invariance under fairness restrictions, the preservation of deadlock-freedom, and the equivalence of busy-wait and wait-until behavior.

An open issue that we study and solve in this paper is whether the π-calculus can be embedded into *CHF* under strong conditions. A strong and well-behaved embedding would increase the knowledge of concurrency and the π-calculus on one hand, and on the other, it leads to an executable implementation with the same semantics

Our translation τ_0 translates closed processes into one single thread which executes a CHF-program. The translation uses one-place buffers (MVars) that can be empty or filled and have a synchronization property: they block on requests to empty an empty buffer, and on filling a full buffer. The translation will use 2 buffers per interaction, since a single buffer is insufficient to simulate the message passing as well as the the synchronization of the π-calculus. We think that the translation is close to being optimal. The translation into Concurrent Haskell has been implemented and tested [Han18]. We succeeded in mathematically confirming that the translation τ_0 has strong and nice semantic properties. These results will also permit to draw the conclusion that the expressiveness of the π-calculus is completely available in Concurrent Haskell.

Our novel *results* are the definition of a translation τ_0 from the π-calculus into CHF, and full abstractness of the translation (Theorem 5.6), and adequacy of the open translation τ (Theorem 5.7). From a technical point of view, a novelty is the comparison of the π-calculus with a concurrent programming language using contextual semantics for may-convergence and should-convergence in both calculi, which is technically involved since the syntactic details of the standard reductions in both calculi have to be analyzed. The adaptation of the adequacy and full abstraction notions (Def. 5.2) for open processes is also novel.

Related work on translations of the π-calculus into programming languages, calculi and logical systems and investigating the properties are [BBP95], where a translation into a graph-rewriting calculus is given and soundness and completeness w.r.t. operational behavior is proved. The article [YRS04] shows a translation and a proof that the π-calculus is exactly operationally represented To the best of our knowledge, there is no deep investigation into semantic properties of a translation of the π-calculus into a programming language w.r.t. contextual semantics.

Outline. After introducing both calculi in Sects. 2 and 3, the translation is defined and explained in Sect. 4. In Sect. 5 we analyze properties of the translation and present our results, while the main part of the proof is given in Sect.6. We conclude in Sect. 7.

2 The π-Calculus with Stop

In this section we explain the π-calculus [MPW92,Mil99,SW01b] in a variant extended with a constant `Stop` [SS15], that signals successful termination of the whole π-calculus program. The π-calculus without the constant `Stop` and with so-called barbed convergences [SW01a] is equivalent w.r.t. semantic properties (see Theorem A.4 in the appendix). Thus, adding the constant `Stop` is not essential, however, the treatment and the translation are easier to explain for the π-calculus with `Stop`. The syntax of processes is as follows:

Definition 2.1. *Let $\mathcal{N}$ be a countable set of (channel) names and $x, y \in \mathcal{N}$. Then processes $P, Q \in \Pi_{\texttt{Stop}}$ are of the form $P, Q \in \Pi_{\texttt{Stop}} := \nu x.P \mid \bar{x}(y).P \mid x(y).P \mid !P \mid P \mid Q \mid 0 \mid \texttt{Stop}$. Free names $FN(P)$, bound names $BN(P)$, and α-equivalence $=_\alpha$ in $\Pi_{\texttt{Stop}}$ are as usual in the π-calculus. Let $\Pi^c_{\texttt{Stop}}$ denote the closed processes in $\Pi_{\texttt{Stop}}$.*

We briefly explain the language constructs. Name restriction $\nu x.P$ restricts the scope of name x to process P, $P \mid Q$ is the parallel composition of P and Q, the process $\bar{x}(y).P$ waits on channel x to output y over channel x and becoming P thereafter, the process $x(y).P$ waits on channel x to receive input, after receiving the input z, the process turns into $P[z/y]$ (where $P[z/y]$ is the substitution of all free occurrences of name y by name z in process P), the process $!P$ denotes the replication of process P, i.e. it behaves like an infinite parallel combination of process P with itself, the process 0 is the silent process that does nothing, and `Stop` is a process constant that signals successful termination.

Definition 2.2. *The only reduction rule of the $\Pi_{\texttt{Stop}}$-calculus is the so-called* interaction $\xrightarrow{ia}$ *which is defined as $x(y).P \mid \bar{x}(z).Q \xrightarrow{ia} P[z/y] \mid Q$.*

Definition 2.3. *Let P, Q, R be processes and x, y channel names, then structural congruence $\equiv$ is the least congruence satisfying the following laws:*

$$\begin{array}{rrr} P \equiv Q, \text{ if } P =_\alpha Q & P \mid 0 \equiv P & \nu x.\texttt{Stop} \equiv \texttt{Stop} \\ P \mid (Q \mid R) \equiv (P \mid Q) \mid R & P \mid Q \equiv Q \mid P & \nu x, y.P \equiv \nu y, x.P \\ \nu x.(P \mid Q) \equiv P \mid \nu x.Q, \text{ if } x \notin FN(P) & \nu x.0 \equiv 0 & !P \equiv P \mid !P \end{array}$$

The communication that sends name y over channel x and then sends u over channel y is: $(x(z).\bar{z}(u).0 \mid \bar{x}(y).y(x).0) \xrightarrow{ia} (\bar{z}(u).0[y/z] \mid y(x).0) \equiv (\bar{y}(u).0 \mid y(x).0) \xrightarrow{ia} (0 \mid 0) \equiv 0$

Definition 2.4. *A* process context *$C \in \mathcal{C}$ is a process that has a a hole $[\cdot]$ at one process position. They are defined by the grammar*

$$C \in \mathcal{C} := [\cdot] \mid \bar{x}(y).C \mid x(y).C \mid C \mid P \mid P \mid C \mid !C \mid \nu x.C, \text{ with } x, y \in \mathcal{N}$$

With $C[P]$ we denote the substitution of the hole in C by process P. Reduction contexts *$PCtxt_\pi$ are defined as $\mathbb{D} \in PCtxt_\pi ::= [\cdot] \mid \mathbb{D} \mid P \mid P \mid \mathbb{D} \mid \nu x.\mathbb{D}$.*

Definition 2.5. *A* standard reduction *$\xrightarrow{sr}$ is the application of $\xrightarrow{ia}$ within a reduction context (modulo structural congruence):*

$$\frac{P \equiv \mathbb{D}[P'], P' \xrightarrow{ia} Q', \mathbb{D}[Q'] \equiv Q, \text{ and } \mathbb{D} \in PCtxt}{P \xrightarrow{sr} Q}$$

Let $\xrightarrow{sr,n}$ denote n standard reductions and $\xrightarrow{sr,*}$ denote the reflexive-transitive closure of $\xrightarrow{sr}$.

Definition 2.6. *A process $P \in \Pi_{\texttt{Stop}}$ is* successful, *if $P \equiv \mathbb{D}[\texttt{Stop}]$ for some $\mathbb{D} \in PCtxt_\pi$.*

We are interested if the standard reduction of a process P successfully terminates. Since reduction is nondeterministic, we define observations which test the existence of a successful sequence (may-convergence), and which test all reduction possibilities (should-convergence):

Definition 2.7. *Let P be a $\Pi_{\mathtt{Stop}}$-process. We say P is* may-convergent *(written $P{\downarrow}$), iff there is a successful process P' with $P \xrightarrow{sr,*} P'$. We say P is* should-convergent *(written $P{\Downarrow}$), iff, for all P': $P \xrightarrow{sr,*} P'$ implies $P'{\downarrow}$. If P is not may-convergent, then P is* must-divergent *(written $P{\Uparrow}$). If P is not should-convergent, then we say it is* may-divergent *(written $P{\uparrow}$).*

Example 2.8. The process $P := \nu x, y.(x(z).0 \mid \bar{x}(y).\mathtt{Stop}))$ is may-convergent ($P{\downarrow}$) and should-convergent ($P{\Downarrow}$), since $P \xrightarrow{sr} 0 \mid \mathtt{Stop}$ is the only standard reduction sequence for P. The process $P' := \nu x, y.(x(z).0 \mid \bar{x}(y).0)$ deterministically reduces to the silent process (i.e. $P' \xrightarrow{sr} 0$), hence it is may-divergent ($P'{\uparrow}$) and even must-divergent ($P'{\Uparrow}$). The process $P'' := \nu x, y.(\bar{x}(y).0 \mid x(z).\mathtt{Stop} \mid x(z).0)$ shows that may-convergence and should-convergence are different. We have $P'' \xrightarrow{sr} \nu x, y.(\mathtt{Stop} \mid x(z).0)$ and $P'' \xrightarrow{sr} \nu x, y.x(z).\mathtt{Stop}$, where the first result is successful, and the second result is not successful. Hence P'' is may-convergent but not should-convergent. It is also may-divergent, but not must-divergent.

Note that should-convergence implies may-convergence, and that must-divergence implies may-divergence. We define general relations between processes:

Definition 2.9. *For processes P and $Q \in \Pi_{\mathtt{Stop}}$ and observation $\xi \in \{{\downarrow}, {\Downarrow}, {\uparrow}, {\Uparrow}\}$, we define $P \leq_\xi Q$ iff $P\xi \implies Q\xi$. The ξ-contextual preorders and ξ-contextual equivalences are defined as $P \leq_{c,\xi} Q$ iff $\forall C \in \mathcal{C} : C[P] \leq_\xi C[Q]$, and $P \sim_{c,\xi} Q$ iff $P \leq_{c,\xi} Q \wedge Q \leq_{c,\xi} P$.* Contextual equivalence *of $\Pi_{\mathtt{Stop}}$-processes is defined as $P \sim_c Q$ iff $P \sim_{c,\downarrow} Q \wedge P \sim_{c,\Downarrow} Q$.*

Since applying structural congruence is highly nondeterministic, and in order to facilitate reasoning on standard reduction sequences, we define a more restrictive use of the congruence laws which is still nondeterministic but only applies the laws on the surface of the process.

Definition 2.10. *Let $\xrightarrow{dsc}$ be the union of the following rules, where $\mathbb{D} \in PCtxt_\pi$:*

(assocl)	$\mathbb{D}[P_1 \mid (P_2 \mid P_3)] \to \mathbb{D}[(P_1 \mid P_2) \mid P_3]$	*(nuup1)*	$\mathbb{D}[(\nu z.P_1) \mid P_2] \to \mathbb{D}[\nu z.(P_1 \mid P_2)]$, *if z does not occur free in P_2.*
(assocr)	$\mathbb{D}[(P_1 \mid P_2) \mid P_3] \to \mathbb{D}[P_1 \mid (P_2 \mid P_3)]$		
(replunfold)	$\mathbb{D}[!P] \to \mathbb{D}[P \mid !P]$	*(nuup2)*	$\mathbb{D}[\nu x.\nu z.P] \to \mathbb{D}[\nu z.\nu x.P]$ *if $x \neq z$*
		(commute)	$\mathbb{D}[P_1 \mid P_2] \to \mathbb{D}[P_2 \mid P_1]$

Let $\xrightarrow{dia}$ be the closure of $\xrightarrow{ia}$ by reduction contexts $PCtxt_\pi$ and let $\xrightarrow{dsr}$ be defined as the composition $\xrightarrow{dsc,*} \cdot \xrightarrow{dia} \cdot \xrightarrow{dsc,*}$.

We omit the proof of the following equivalences, but it can be constructed completely analogous to the proof given in [Sab14] for barbed may- and should-testing.

Theorem 2.11. *For all processes $P \in \Pi_{\mathtt{Stop}}$ the following holds: i) $P{\downarrow}$ iff $P \xrightarrow{dsr,*} \mathbb{D}[\mathtt{Stop}]$ ii) $P{\uparrow}$ iff $\exists P'$ such that $P \xrightarrow{dsr,*} P'$ and $P'{\Uparrow}$.*

The previous theorem allows us to restrict standard reduction to $\xrightarrow{dsr}$-reduction when reasoning on reduction sequences that witness may-convergence or may-divergence, resp.

3 The Process Calculus *CHF*

We recall the program calculus *CHF* [SSS11,SSS12] which models a core language of Concurrent Haskell extended by futures. We assume a partitioned set of *data constructors* c such that each family represents a type T and such that the data constructors of type T are $c_{T,1}, \ldots, c_{T,|T|}$ where each $c_{T,i}$ has an arity $\mathrm{ar}(c_{T,i}) \geq 0$. The two-layered syntax of the calculus *CHF* has

$$
\begin{array}{rl}
P \in \mathit{Proc} ::= & (P_1 \mid P_2) \mid x \Leftarrow e \mid \nu x.P \mid x\,\mathbf{m}\,e \mid x\,\mathbf{m}- \mid x = e \\
e \in \mathit{Expr} ::= & x \mid \lambda x.e \mid (e_1\ e_2) \mid \texttt{seq}\ e_1\ e_2 \mid c\ e_1 \ldots e_{\mathrm{ar}(c)} \mid \texttt{letrec}\ x_1{=}e_1, \ldots, x_n{=}e_n\ \texttt{in}\ e \\
& \mid \texttt{case}_T\ e\ \texttt{of}\ (c_{T,1}\ x_1 \ldots x_{\mathrm{ar}(c_{T,1})}\ \texttt{->}\ e_1) \ldots (c_{T,|T|}\ x_1 \ldots x_{\mathrm{ar}(c_{T,|T|})}\ \texttt{->}\ e_{|T|}) \mid m \\
m \in \mathit{MExpr} ::= & \texttt{return}\,e \mid e \gg\!= e' \mid \texttt{future}\,e \mid \texttt{takeMVar}\,e \mid \texttt{newMVar}\,e \mid \texttt{putMVar}\,e\,e' \\
\mathsf{t} \in \mathit{Typ} ::= & \texttt{IO}\ \mathsf{t} \mid (T\ \mathsf{t}_1 \ldots \mathsf{t}_n) \mid \texttt{MVar}\ \mathsf{t} \mid \mathsf{t}_1 \to \mathsf{t}_2
\end{array}
$$

Fig. 1. Syntax of expressions, processes, and types of *CHF*

$$
\begin{array}{l}
P_1 \mid P_2 \equiv P_2 \mid P_1 \\
(P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3) \\
(\nu x.P_1) \mid P_2 \equiv \nu x.(P_1 \mid P_2) \text{ if } x \notin FV(P_2) \\
\nu x_1.\nu x_2.P \equiv \nu x_2.\nu x_1.P \\
P_1 \equiv P_2 \text{ if } P_1 =_\alpha P_2
\end{array}
$$

Fig. 2. Structural congruence of *CHF*

$$
\begin{array}{rl}
\mathbb{D} \in \mathit{PCtxt} ::= & [\cdot] \mid \mathbb{D} \mid P \mid P \mid \mathbb{D} \mid \nu x.\mathbb{D} \\
\mathbb{M} \in \mathit{MCtxt} ::= & [\cdot] \mid \mathbb{M} \gg\!= e \\
\mathbb{F} \in \mathit{FCtxt} ::= & \mathbb{E} \mid (\texttt{takeMVar}\,\mathbb{E}) \mid (\texttt{putMVar}\,\mathbb{E}\,e) \\
\mathbb{E} \in \mathit{ECtxt} ::= & [\cdot] \mid (\mathbb{E}\ e) \mid (\texttt{seq}\ \mathbb{E}\ e) \\
& \mid (\texttt{case}\ \mathbb{E}\ \texttt{of}\ \mathit{alts})
\end{array}
$$

Fig. 3. *PCtxt*-, *MCtxt*-, *FCtxt*-, *ECtxt*-Contexts

Monadic Computations

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

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

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

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

(fork) $y \Leftarrow \mathbb{M}[\texttt{future}\ e] \xrightarrow{sr} \nu z.(y \Leftarrow \mathbb{M}[\texttt{return}\ z] \mid z \Leftarrow e)$ where z is fresh

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

Functional Evaluation

(cpce) $y \Leftarrow \mathbb{M}[\mathbb{F}[x]] \mid x = e \xrightarrow{sr} y \Leftarrow \mathbb{M}[\mathbb{F}[e]] \mid x = e$

(mkbinds) $y \Leftarrow \mathbb{M}[\mathbb{F}[\texttt{letrec}\ x_1{=}e_1, \ldots, x_n{=}e_n\ \texttt{in}\ e]]$
$\xrightarrow{sr} \nu x_1 \ldots x_n.(y \Leftarrow \mathbb{M}[\mathbb{F}[e]] \mid x_1{=}e_1 \mid \ldots \mid x_n{=}e_n)$

(beta) $y \Leftarrow \mathbb{M}[\mathbb{F}[((\lambda x.e_1)\ e_2)]] \xrightarrow{sr} y \Leftarrow \mathbb{M}[\mathbb{F}[e_1[e_2/x]]]$

(case) $y \Leftarrow \mathbb{M}[\mathbb{F}[\texttt{case}_T\ (c\ e_1 \ldots e_n)\ \texttt{of}\ \ldots (c\ y_1 \ldots y_n\ \texttt{->}\ e) \ldots]]$
$\xrightarrow{sr} y \Leftarrow \mathbb{M}[\mathbb{F}[e[e_1/y_1, \ldots, e_n/y_n]]]$

(seq) $y \Leftarrow \mathbb{M}[\mathbb{F}[(\texttt{seq}\ v\ e)]] \xrightarrow{sr} y \Leftarrow \mathbb{M}[\mathbb{F}[e]]$ where v is a functional value

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

Capture avoidance: We assume capture avoiding reduction for all reduction rules.

Fig. 4. Standard reduction rules of *CHF* (call-by-name-version)

processes $P \in \mathit{Proc}$ on the top-layer which may have expressions $e \in \mathit{Expr}$ (the second layer) as subterms. Processes and expressions are defined by the grammars in Fig. 1 where u, w, x, y, z denote variables from a countably-infinite set of variables *Var*. As in the $\Pi_{\mathtt{Stop}}$-calculus parallel processes are formed by parallel composition " $\mid$ ". The ν-binder restricts the scope of a variable. A concurrent thread $x \Leftarrow e$ evaluates the expression e and binds the result to the variable x (called the *future* x). In a process there is (at most one) unique distinguished thread, called the *main thread* written as $x \overset{\mathsf{main}}{\Longleftarrow} e$. MVars are mutable variables which are empty or filled. A thread blocks if it wants to fill a filled MVar $x\,\mathbf{m}\,e$ or empty an empty MVar $x\,\mathbf{m}-$. The variable x is called the *name of the MVar*. Bindings $x = e$ model the global heap, where x is called a *binding variable*. For a process P, a variable x is an *introduced variable* if x is a future, a name of an MVar, or a binding variable. An introduced variable is visible to the whole process unless its scope is restricted by a ν-binder, i.e. in $Q \mid \nu x.P$ the scope of x is P. A process is *well-formed*, if all introduced variables are pairwise distinct, and there exists at most one main thread $x \overset{\mathsf{main}}{\Longleftarrow} e$.

Expressions *Expr* consist of functional expressions and monadic expressions *MExpr* which model IO-operations. Functional expressions contain variables, *abstractions* $\lambda x.e$, *applications* $(e_1\ e_2)$, *constructor applications* $(c\ e_1\ \ldots\ e_{\mathrm{ar}(c)})$, `letrec`-*expressions* $(\texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e)$, $\texttt{case}_T$-*expressions* for every type T, and `seq`-*expressions* $(\texttt{seq}\ e_1\ e_2)$. We abbreviate `case`-expressions as $\texttt{case}_T\ e\ \texttt{of}\ \mathit{alts}$ where *alts* are the `case`-*alternatives*. The `case`-alternatives must have exactly one alternative $(c_{T,i}\ x_1 \ldots x_{\mathrm{ar}(c_{T,i})}\ \texttt{->}\ e_i)$ for every constructor $c_{T,i}$ of type T, where the variables $x_1, \ldots, x_{\mathrm{ar}(c_{T,i})}$ (occurring in the *pattern* $c_{T,i}\ x_1 \ldots x_{\mathrm{ar}(c_{T,i})}$) are pairwise distinct and become bound with scope e_i.

In (`letrec` $x_1 = e_1, \ldots, x_n = e_n$ `in` e) the variables $x_1, \ldots, x_n$ are pairwise distinct and the bindings $x_i = e_i$ are recursive, i.e. the scope of x_i is $e_1, \ldots, e_n$ and e. *Monadic operators* `newMVar`, `takeMVar`, and `putMVar` are used to create and access MVars, the "bind"-operator `>>=` implements the sequential composition of IO-operations, the `future`-operator performs thread creation, and `return` lifts expressions to monadic expressions. *Functional values* are abstractions and constructor applications. A monadic expression of the form (`return` e), (e_1 `>>=` e_2), (`future` e), (`newMVar` e), (`takeMVar` e), or (`putMVar` $e_1\, e_2$) is called a *monadic value*. A *value* is either a functional value or a monadic value.

Variable binders are introduced by abstractions, `letrec`-expressions, `case`-alternatives, and $\nu x.P$. This induces free and bound variables, α-renaming, and α-equivalence $=_\alpha$. Let $FV(P)$ ($FV(e)$, resp.) be the free variables of process P (expression e, resp.). We assume the *distinct variable convention* to hold: free variables are distinct from bound variables, and bound variables are pairwise distinct. We assume that reductions perform α-renaming to obey this convention. In Fig. 2 structural congruence $\equiv$ of *CHF*-processes is defined.

We assume that every expression and process is well-typed according to a monomorphic type system. The syntax of types is in Fig. 1 where (`IO` $\mathfrak{t}$) stands for a monadic action with return type $\mathfrak{t}$, (`MVar` $\mathfrak{t}$) stands for an `MVar` with content type $\mathfrak{t}$, and $\mathfrak{t}_1 \to \mathfrak{t}_2$ is a function type. Since the type system is standard, we omit the details, but they can be found in [SSS11].

We recall the call-by-name small-step reduction for *CHF* (which is semantically equivalent to a call-by-need semantics, see [SSS11]). A context is a process or an expression with a (typed) hole $[\cdot]$. We introduce some classes of contexts in Fig. 3. On the process level there are *process contexts PCtxt*, on expressions first *monadic contexts MCtxt* are used to find the next to-be-evaluated monadic action in a sequence of actions. For the evaluation of (purely functional) expressions, usual (call-by-name) *expression evaluation contexts* $\mathbb{E} \in ECtxt$ are used, and to enforce the evaluation of the (first) argument of the monadic operators `takeMVar` and `putMVar` the class of *forcing contexts* $\mathbb{F} \in FCtxt$ is used. A *functional value* is an abstraction or a constructor application, a *value* is a functional value or a monadic expression in *MExpr*.

Definition 3.1. *The call-by-name standard reduction* $\xrightarrow{sr}$ *is defined by the rules and the closure in Fig. 4. We permit standard reduction only for well-formed processes.*

Functional evaluation includes call-by-name β-reduction (beta), a rule (cpce) for copying shared bindings into a needed position, rules (case) and (seq) to evaluate `case`- and `seq`-expressions, and rule (mkbinds) to move `letrec`-bindings into the global set of shared bindings. For monadic computations, rule (lunit) implements the monad by applying the first monad law to proceed a sequence of actions. Rules (nmvar), (tmvar), and (pmvar) handle the MVar creation and access. A `takeMVar`-operation can only be performed on a filled MVar, and a `putMVar`-operation needs an empty MVar for being executed. Rule (fork) spawns a new concurrent thread, where the calling thread receives the name of the thread (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)).

A well-formed process P is *successful*, if $P \equiv \nu x_1 \ldots \nu x_n.(x \overset{\mathsf{main}}{\Longleftarrow}$ `return` $e \mid P')$. These are the desired results of standard reduction sequences. The successful processes capture the behavior that termination of the main-thread implies termination of the whole program. We permit standard reductions only for well-formed processes which are not successful.

Definition 3.2. *Let P be a CHF-process. Process P* may-converges *(written as $P{\downarrow}$), iff it is well-formed and reduces to a successful process, i.e. $P{\downarrow}$ iff P is well-formed and $\exists P' : P \xrightarrow{sr,*} P' \wedge P'$ successful. If $P{\downarrow}$ does not hold, then P* must-diverges *written as $P{\Uparrow}$. Process P* should-converges *(written as $P{\Downarrow}$), iff it is well-formed and remains may-convergent under reduction, i.e. $P{\Downarrow}$ iff P is well-formed and $\forall P' : P \xrightarrow{sr,*} P' \implies P'{\downarrow}$ If P is not should-convergent then we say P* may-diverges *written as $P{\uparrow}$.*

Note that a process P is may-divergent if there is a finite reduction sequence $P \xrightarrow{sr,*} P'$ such that $P'\Uparrow$. Definition 3.2 implies that non-well-formed processes are always must-divergent, since they are irreducible and never successful.

Definition 3.3. *Contextual approximation $\leq_c$ and contextual equivalence $\sim_c$ on CHF-processes are defined as $\leq_c := \leq_{c,\downarrow} \cap \leq_{c,\Downarrow}$ and $\sim_c := \leq_c \cap \geq_c$ where*

- $P_1 \leq_{c,\downarrow} P_2$ *iff* $\forall \mathbb{D} \in PCtxt : \mathbb{D}[P_1]\downarrow \implies \mathbb{D}[P_2]\downarrow$
- $P_1 \leq_{c,\Downarrow} P_2$ *iff* $\forall \mathbb{D} \in PCtxt : \mathbb{D}[P_1]\Downarrow \implies \mathbb{D}[P_2]\Downarrow$

*For CHF-*expressions*, let $e_1 \leq_c e_2$ iff for all process-contexts C with a hole at expression position: $C[e_1] \leq_c C[e_2]$ and $e_1 \sim_c e_2$ iff $e_1 \leq_c e_2 \wedge e_2 \leq_c e_1$.*

Proposition 3.4 ([SSS11]). *Let P_1, P_2 be well-formed processes with $P_1 \equiv P_2$. Then $P_1 \sim_c P_2$.*

We require several correct program transformations for later reasoning. A program transformation T is a binary relation between *CHF*-processes.

Definition 3.5. *A program transformation T is called* correct*, iff $e_1\ T\ e_2 \implies e_1 \sim_c e_2$.*

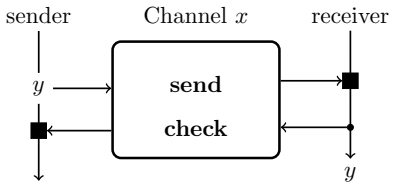
Deterministic variants of (tmvar) and (pmvar) are (dtmvar), (dpmvar), respectively.

Proposition 3.6 ([SSS11]). *The reduction rules (lunit), (nmvar), (fork), (unIO), (cpce), (mkbinds), (beta), (case), (seq) (dtmvar), and (dpmvar) are correct program transformations, whereas (pmvar) and (tmvar) in general are not correct as program transformations.*

4 The Translation

We define a translation τ_0 mapping $\Pi_{\texttt{Stop}}$-processes to *CHF*-processes. It uses the data type `Channel`, defined in Haskell-syntax as `data Channel = Channel (MVar Channel) (MVar ())`. `Channel` is a recursive data type, which can be initialized with a $\bot$-expression. In the following, we abbreviate `case` e `of (Channel` *sendx checkx* `->` *sendx*`)` as `getsend` e and `case` e `of (Channel` *sendx checkx* `->` *checkx*`)` as `getcheck` e. We use $a \gg b$ as abbreviation for $a \gg\!= (\lambda\_ . b)$ and also use Haskell's do-notation as abbrevation, where $\mathbf{do}\ \{x \leftarrow a; b\} = a \gg\!= \lambda x.(\mathbf{do}\ b)$, $\mathbf{do}\ \{a; b\} = a \gg (\mathbf{do}\ b)$, and $\mathbf{do}\ \{\texttt{return}\ e\} = e$.

The translation from the $\Pi_{\texttt{Stop}}$-calculus is done by creating two MVars per channel, the first one contains the (translated name) of the channel and the second is for the synchronization.



Message y is sent over channel x. Initially, the MVars *send* and *check* are empty. The receiver waits (black box) for a message on *send*, until the receiver fills it with y. Now the sender waits until *check* is filled, and the receiver acknowledges by putting () into MVar *check*.

Definition 4.1. *We define the translation τ_0 and its inner translation τ from the $\Pi_{\texttt{Stop}}$-calculus into the CHF-calculus as follows, using the* **do***-notation.*

$$
\begin{array}{lll}
\tau_0(P) & = z \overset{\text{main}}{\Longleftarrow} \mathbf{do}\ \{\mathit{stop} \leftarrow \texttt{newMVar}\ (); \texttt{future}\ \tau(P); \texttt{putMVar}\ \mathit{stop}\ ()\} \\
 & \quad \textit{where the process } P \textit{ must be closed} \\
\tau(\bar{x}(y).P) & = \mathbf{do}\ \{\texttt{putMVar}\ (\texttt{getsend}\ x)\ y; \texttt{takeMVar}\ (\texttt{getcheck}\ x); \tau(P)\} \\
\tau(x(y).P) & = \mathbf{do}\ \{y \leftarrow \texttt{takeMVar}\ (\texttt{getsend}\ x); \texttt{putMVar}\ (\texttt{getcheck}\ x)\ (); \tau(P)\} \\
\tau(P \mid Q) & = \mathbf{do}\ \{\texttt{future}\ \tau(Q); \tau(P)\} \\
\tau(\nu x.P) & = \mathbf{do}\ \{\mathit{sendx} \leftarrow \texttt{newMVar}\ \bot; \mathit{checkx} \leftarrow \texttt{newMVar}\ (); \texttt{takeMVar}\ \mathit{sendx}; \\
 & \qquad \texttt{takeMVar}\ \mathit{checkx}; \texttt{letrec}\ x = \texttt{Channel}\ \mathit{sendx}\ \mathit{checkx}\ \texttt{in}\ \tau(P)\} \\
\tau(0) & = \texttt{return}\ () \\
\tau(\texttt{Stop}) & = \texttt{takeMVar}\ \mathit{stop} \\
\tau(!P) & = \texttt{letrec}\ f = \mathbf{do}\ \{\texttt{future}\ \tau(P); f\}\ \texttt{in}\ f
\end{array}
$$

Note that the futures (as a name of process-results) are not required for the translation. The translation τ_0 generates a main-thread and an MVar *stop*. The main thread is then waiting for the MVar *stop* to be emptied. The inner translation τ translates the constructs and constants of the $\Pi_{\texttt{Stop}}$-calculus into *CHF*-expressions.

Remark 4.2. Note that a translation, without a control like the check-MVar, would produce unexpected effects. Consider the translation ρ_0 that is like τ_0, (and ρ like τ), but only uses the send-MVar, and omits the check-MVar. Consider the process $P = \nu x.\bar{x}y.x(z).\texttt{Stop}$ that is stuck in the π-calculus and thus must-divergent. The translation ρ of P is: $\rho(P) = \mathbf{do}\ldots;\texttt{putMVar}\ (\texttt{getsend}\ x)\ y;\texttt{takeMVar}\ ($ which first initializes the MVar `sendx` to be empty, and then makes a put and a get, and then reduces to a successful process in CHF, and hence $\rho_0(P)$ is may-convergent. Thus ρ_0 fails as a translation.

The translation τ_0 does not make use of the name of threads / futures. We use this to simplify the notation by omitting the name of the future and its ν-binder, and notationally, abbreviating the construct $x \Leftarrow e$ by writing $\Leftarrow e$. This leads to the following conventions. In the usual notation, $x \Leftarrow ((\texttt{future}\ e) \gg e')$ reduces (in two steps) to $\nu z.z \Leftarrow e \mid x \Leftarrow e'$. In the simplified notation, $\Leftarrow(\texttt{future}\ e \gg e')$ reduces to $\Leftarrow e \mid \Leftarrow e'$.

As a further example, $\Leftarrow \tau(!P)$ is $\Leftarrow \texttt{letrec}\ f = \mathbf{do}\ \{\texttt{future}\ \tau(P); f\}\ \texttt{in}\ f$, which sr-reduces to $\nu f.(\Leftarrow f \mid f = \mathbf{do}\ \{\texttt{future}\ \tau(P); f\})$, which reduces further to the process $\nu f.(\Leftarrow \mathbf{do}\ \{\texttt{future}\ \tau(P); f\} \mid f = \mathbf{do}\ \{\texttt{future}\ \tau(P); f\})$ and then it reduces to the process $\nu f.(\Leftarrow f \mid \Leftarrow \tau(P) \mid f = \mathbf{do}\ \{\texttt{future}\ \tau(P); f\})$.

Remark 4.3. Note that (except for the main-thread) the translation τ_0 generates a Concurrent Haskell-program, i.e. if we write $\tau_0(P) = z \overset{\mathsf{main}}{\Longleftarrow} e$ as $\texttt{main} = e$, we can execute the translation in the Haskell-interpreter. Here we assume that the `future`-primitive is implemented as suggested in [SSS11]. An alternative is to replace all expressions $(\texttt{future}\ e)$ by $(\texttt{forkIO}\ e)$ and changing the type of these expressions from `IO t` to `IO ()`. This is possible, since the translated program never uses the resulting futures generated by `future`.

Note that the translation σ_0, which will be defined below, does not generate a Haskell program, since it maps to process components.

5 Properties of the Translation

In this section we define properties of translations and present our results for the translations τ_0 and τ. We defer the proof of convergence-equivalence to Sect. 6.

Definition 5.1. *The context C^{τ}_{out} for the translation τ_0 is defined as*

$$C^{\tau}_{out} = \nu f, stop.z \overset{\mathsf{main}}{\Longleftarrow} \mathbf{do}\ \{stop \leftarrow \texttt{newMVar}\ ();\texttt{future}\ [\cdot];\texttt{putMVar}\ stop\ ()\}$$

We define the relations $\leq_{c,\tau.0}$ and $\sim_{c,\tau.0}$ on CHF-expressions as follows:

$$\begin{array}{l} e_1 \leq_{c,\tau.0} e_2 \text{ iff } \forall C \text{ such that } C^{\tau}_{out}[C[e_1]], C^{\tau}_{out}[C[e_2]] \text{ are closed }: \\ \qquad\qquad C^{\tau}_{out}[C[e_1]]{\downarrow} \implies C^{\tau}_{out}[C[e_2]]{\downarrow} \text{ and } C^{\tau}_{out}[C[e_1]]{\Downarrow} \implies C^{\tau}_{out}[C[e_2]]{\Downarrow} \\ e_1 \sim_{c,\tau.0} e_2 \text{ iff } e_1 \leq_{c,\tau.0} e_2 \text{ and } e_1 \leq_{c,\tau.0} e_2. \end{array}$$

For the numbered items 4,5,6 below, we adapt the view of the translation τ to a translation from $\Pi_{\texttt{Stop}}$ into a slightly modified *CHF*-language, without an explicit initialization and where evaluations take place in the context C^{τ}_{out}. Let $\Pi^{c}_{\texttt{Stop},C}$ be the closed contexts of $\Pi_{\texttt{Stop}}$.

Definition 5.2. *The following properties of τ_0 and τ are essential (see [SSNSS08,SSNS15]) .*

1. $\forall P, P' \in \Pi^c_{\mathtt{Stop}}: \quad \tau_0(P) \sim_c \tau_0(P') \implies P \sim_c P'$ *(closed-adequacy of τ_0)*
2. $\forall P, P' \in \Pi^c_{\mathtt{Stop}}: \quad P \sim_c P' \iff \tau_0(P) \sim_c \tau_0(P')$ *(closed-full abstraction of τ_0)*
3. $\forall P, P' \in \Pi^c_{\mathtt{Stop}}: \quad P{\downarrow} \iff \tau_0(P){\downarrow}$ *and* $P{\Downarrow} \iff \tau_0(P){\Downarrow}$ *(convergence-equivalence of τ_0)*
4. $\forall P \in \Pi_{\mathtt{Stop}}: \forall C \in \Pi^c_{\mathtt{Stop},C}: \forall \xi \in \{{\downarrow}, {\Downarrow}\}:$
   $C^\tau_{out}[\tau(C[P])]\xi \iff C^\tau_{out}[\tau(C)[\tau(P)]]\xi$ *(compositionality of τ)*
5. $\forall P, P' \in \Pi_{\mathtt{Stop}}: \tau(P) \sim_{c,\tau.0} \tau(P') \implies P \sim_c P'$ *(adequacy of τ)*
6. $\forall P, P' \in \Pi_{\mathtt{Stop}}: \quad P \sim_c P' \iff \tau(P) \sim_{c,\tau.0} \tau(P')$ *(full abstraction of τ)*

First we show a simple form of a context lemma:

Lemma 5.3. *Let e, e' be CHF-expressions, where the only free variable is stop. Then $C^\tau_{out}[e] \sim_c C^\tau_{out}[e']$ iff $C^\tau_{out}[e]{\downarrow} \iff C^\tau_{out}[e']{\downarrow}$ and $C^\tau_{out}[e]{\Downarrow} \iff C^\tau_{out}[e']{\Downarrow}$.*

Proof. One direction is obvious by using the empty context in the $\sim_c$-definition. For the other direction, assume $C^\tau_{out}[e]{\downarrow} \iff C^\tau_{out}[e']{\downarrow}$ and $C^\tau_{out}[e]{\Downarrow} \iff C^\tau_{out}[e']{\Downarrow}$. There are two cases:

1. Let $\mathbb{D}$ be a process-context and $\mathbb{D}[C^\tau_{out}[e]]{\downarrow}$. Then we have to show that $\mathbb{D}[C^\tau_{out}[e']]{\downarrow}$. Due to closedness, $\mathbb{D}[C^\tau_{out}[e]] \equiv P_{\mathbb{D}} \mid C^\tau_{out}[e]$ and $\mathbb{D}[C^\tau_{out}[e']] \equiv P_{\mathbb{D}} \mid C^\tau_{out}[e']$ for some closed *CHF*-process $P_{\mathbb{D}}$. Due to closedness, there is no interference between $P_{\mathbb{D}}$ and $C^\tau_{out}[e']$, or $C^\tau_{out}[e]$, resp. Hence $C^\tau_{out}[e]{\downarrow} \implies C^\tau_{out}[e']{\downarrow}$, and thus $\mathbb{D}[C^\tau_{out}[e']]{\downarrow}$.
2. Let $\mathbb{D}[C^\tau_{out}[e]]{\uparrow}$, we have to show $\mathbb{D}[C^\tau_{out}[e']]{\uparrow}$. Due to closedness, $\mathbb{D}[C^\tau_{out}[e]] \equiv P_{\mathbb{D}} \mid C^\tau_{out}[e]$ and $\mathbb{D}[C^\tau_{out}[e']] \equiv P_{\mathbb{D}} \mid C^\tau_{out}[Q']$ and the same arguments as in item 1 show the claim. □

Proposition 5.4. *The translation τ is compositional.*

Proof. Compositionality follows by checking whether the single cases of the translation τ are independent of the surrounding context, and translate every level independently.

We state our main result, which will be proved in the subsequent section:

Theorem 5.5. *Let $P \in \Pi_{\mathtt{Stop}}$ be closed. Then τ_0 is convergence-equivalent for ${\downarrow}$ and ${\Downarrow}$, i.e. $P{\downarrow}$ is equivalent to $\tau_0(P){\downarrow}$. and $P{\Downarrow}$ is equivalent to $\tau_0(P){\Downarrow}$.*

Proof. This follows from Proposition 6.9 for may-convergence and from Proposition 6.12 for should-convergence. Both propositions will be proved in Sect. 6

Theorem 5.6. *The translation τ_0 is an embedding, i.e. it is closed-adequate and closed-fully abstract: for closed $P_1, P_2 \in \Pi_{\mathtt{Stop}}$, the relation $\tau_0(P_1) \sim_c \tau_0(P_2)$ holds iff $P_1 \sim_c P_2$.*

Proof. The implication $P \sim_c P' \implies \tau_0(P) \sim_c \tau_0(P')$ follows from Lemma 5.3, since τ_0 produces closed processes that are in context C_{out}. For the other direction (closed-adequacy of τ_0), we additionally require that for all closed $\Pi_{\mathtt{Stop}}$-processes P, P', contexts C, and $\xi \in \{{\downarrow}, {\Downarrow}\}$ the implication $(P\xi \iff P'\xi) \implies (C[P]\xi \iff C[P']\xi)$ holds. This can be proved by standard methods and the fact, that a closed $\Pi_{\mathtt{Stop}}$-process cannot communicate with other processes, even if it is replicated.

We show in the following that the translation τ transports $\Pi_{\mathtt{Stop}}$-processes into *CHF*, such that adequacy holds. This is a stronger statement than the full abstraction for closed processes, since this shows that the translated processes also mimic the behaviour of the original $\Pi_{\mathtt{Stop}}$-processes when plugged into contexts in a correct way. Also, the image of $\Pi_{\mathtt{Stop}}$ within *CHF* behaves much the same as the original $\Pi_{\mathtt{Stop}}$-processes.

However, this open translation is not fully abstract, which means that there are be *CHF*-contexts that can see and exploit too much of the details of the translation.

Theorem 5.7. *The translation τ is adequate.*

Proof. Let P, P' be $\Pi_{\texttt{Stop}}$-processes, such that $\tau(P) \leq_{c,\tau,0} \tau(P')$. We show that $P \leq_c P'$. Let C be a context in $\Pi_{\texttt{Stop}}$, such that $C[P]\downarrow$. Then $\tau_0(C[P]) = C^\tau_{out}[\tau(C[P])]$. Closed convergence equivalence shows that $C^\tau_{out}[\tau(C[P])]\downarrow$, which is the same as $C^\tau_{out}[\tau(C)[\tau(P)]]\downarrow$ by Proposition 5.4. The assumption $\tau(P) \leq_{c,\tau,0} \tau(P')$ implies $C^\tau_{out}[\tau(C)[\tau(P')]]\downarrow$, which again is the same as $C^\tau_{out}[\tau(C[P'])]\downarrow$ using Proposition 5.4. Again, closed convergence equivalence implies $C[P']\downarrow$. The same arguments holds for $\Downarrow$ instead of $\downarrow$. In summary, we obtain $P \leq_c P'$, since the computation is possible for every $\Pi_{\texttt{Stop}}$-context C.

Theorem 5.8. *The translation τ is not fully abstract.*

Proof. This holds, since an open translation can be closed in *CHF* by a context without initializing the ν-bound MVars. For $P = \bar{x}y.\texttt{Stop} \mid x(z).\texttt{Stop}$, we have $P \sim_c \texttt{Stop}$ in the π-calculus (see Example A.2), but in *CHF* $\tau(P) \not\sim_c \tau(\texttt{Stop})$: for a process context $\mathbb{D}$ that does not initialize the MVars for x (as the translation does), we have $\mathbb{D}[P]\Uparrow$, but $\mathbb{D}[\texttt{Stop}]\Downarrow$.

6 Proofs of the Convergence Properties of the Translation

6.1 A Top-Down Translation σ

We define a variant σ of the translation τ that generates *CHF*-processes and that is closer to a direct implementation. It refers to τ for the translations parts that generate expressions. As a further abbreviation let us write $\emptyset$ for the (useless) binding $\nu x.x = ()$.

Definition 6.1. *The translation σ_0 is similar to τ_0, but refers to σ instead of τ, and generates processes. The translation σ also generates processes.*

$$
\begin{array}{lll}
\sigma_0(P) & = \nu stop.(\ z \overset{\mathsf{main}}{\Longleftarrow} \texttt{putMVar}\ stop\ ()\ \mid\ stop\,\mathbf{m}\,()\ \mid\ \sigma(P)) & \\
\sigma(\nu x.P) & = \nu x, sendx, checkx.(sendx\,\mathbf{m} - \mid checkx\,\mathbf{m} - \mid x = \texttt{Channel}\ sendx\ checkx \mid \sigma(P)) & \\
\sigma(\bar{x}(y).P) & = \Leftarrow \tau(\bar{x}(y).P) & \sigma(0) = \emptyset \\
\sigma(x(y).P) & = \Leftarrow \tau(x(y).P) & \sigma(\texttt{Stop}) = \Leftarrow \texttt{takeMVar}\ stop \\
\sigma(P \mid Q) & = \sigma(P) \mid \sigma(Q) & \sigma(!P) = \nu f.(\Leftarrow f \mid f = \mathbf{do}\ \{\texttt{future}\ \tau(P); f\})
\end{array}
$$

Definition 6.2. *The context C^σ_{out} for the translation σ_0 is defined as*

$$C^\sigma_{out} = \nu f, stop.(\ z \overset{\mathsf{main}}{\Longleftarrow} \texttt{putMVar}\ stop\ ()\ \mid\ stop\,\mathbf{m}\,()\ \mid\ f \Leftarrow [\cdot]).$$

Lemma 6.3. *For all $\Pi_{\texttt{Stop}}$-processes P: 1. $\tau_0(P) \xrightarrow{sr,*} \sigma_0(P)$ and 2. $\Leftarrow \tau(P) \xrightarrow{sr,*} \sigma(P)$.*

6.2 The Translation Preserves May-Convergence

The goal of this section is to show that may-convergence of a closed $\Pi_{\texttt{Stop}}$-process P implies may-convergence of the translated process $\tau_0(P)$. The main part is to show that for a single step $P \xrightarrow{dia} P'$ or $P \xrightarrow{dsc} P'$ for closed $\Pi_{\texttt{Stop}}$-processes P, P' that there are natural corresponding reduction steps of $\sigma_0(P)$ in the *CHF*-calculus.

Lemma 6.4. *Let $P \in \Pi_{\texttt{Stop}}$ be a closed process, such that $P \xrightarrow{dia} P'$. Then there is a standard reduction sequence $\sigma_0(P) \xrightarrow{sr,*} \sigma_0(P')$ in the CHF-calculus.*

Proof. Let $P = \mathbb{D}[x(y).P_r \mid \bar{x}(z).P_s]$ and $P' = \mathbb{D}[P_r[z/y] \mid P_s]$. Since P is closed, there is a binder νx in $\mathbb{D}$, i.e. we can assume that $\mathbb{D} = \mathbb{D}_1[\nu x.\mathbb{D}_2[\cdot]]$ for some $\mathbb{D}_\pi$-contexts $\mathbb{D}_1, \mathbb{D}_2$. Since $\mathbb{D}_\pi$-contexts have no input- or output-prefix on the hole-path, this shows:

$$
\begin{array}{rl}
\sigma_0(P) = & C^\sigma_{out}[D_1[\nu x, sendx, checkx.sendx\,\mathbf{m} - \mid checkx\,\mathbf{m} - \mid x = \texttt{Channel}\ sendx\ checkx \\
& \mid D_2[\Leftarrow \mathbf{do}\ \{\texttt{putMVar}\ (\texttt{getsend}\ x)\ z; \texttt{takeMVar}\ (\texttt{getcheck}\ x); \tau(P_s)\} \\
& \quad \mid \Leftarrow \mathbf{do}\ \{y \leftarrow \texttt{takeMVar}\ (\texttt{getsend}\ x); \texttt{putMVar}\ (\texttt{getcheck}\ x)\ (); \tau(P_r)\}]]]
\end{array}
$$

where D_1, D_2 are the σ-translations of $\mathbb{D}_1, \mathbb{D}_2$. Inspecting the translation σ shows, that D_1, D_2 are $\mathbb{D}$-contexts. Using a sequence of $\xrightarrow{sr}$-reductions we get (see Appendix C)

$$\sigma_0(P) \xrightarrow{sr,16} Q = C^{\sigma}_{out}[D_1[\nu x, sendx, checkx.sendx\,\mathbf{m} - \mid checkx\,\mathbf{m} - \\ \mid x{=}\texttt{Channel}\ sendx\ checkx \mid D_2[\Leftarrow \tau(P_s) \mid \Leftarrow \tau(P_r)[z/y]]]]$$

The equation $\sigma_0(P') = C^{\sigma}_{out}[D_1[\ \nu x, sendx, checkx.sendx\,\mathbf{m} - \mid checkx\,\mathbf{m} - \\ \mid x = \texttt{Channel}\ sendx\ checkx \mid D_2[\sigma(P_r[z/y]) \mid \sigma(P_s)]]]$
and Lemma 6.3 imply $Q \xrightarrow{sr,*} \sigma_0(P')$.

Lemma 6.5. *Let $P, P' \in \Pi_{\texttt{Stop}}$ be closed, such that $P \xrightarrow{dsc} P'$. Then $\sigma_0(P) \xrightarrow{sr,*} \sigma_0(P')$, if rule (replunfold) is used, and $\sigma_0(P) \equiv \sigma_0(P')$, otherwise. In particular, $\sigma_0(P) \sim_c \sigma_0(P')$.*

Proof. For rules (assocl),(assocr), we have $\sigma_0(\mathbb{D}[(P_1 \mid P_2) \mid P_3]) = C[(\sigma(P_1) \mid \sigma(P_2)) \mid \sigma(P_3)] \equiv C[\sigma(P_1) \mid (\sigma(P_2) \mid \sigma(P_3))] = \sigma_0(\mathbb{D}[P_1 \mid (P_2 \mid P_3)])$ for some *CHF* $\mathbb{D}$-context C. Hence, we have $\sigma_0(P) \equiv \sigma_0(P')$.

For the rule (nuup1), we have $\sigma_0(\mathbb{D}[(\nu z.P_1) \mid P_2]) = C[\sigma(\nu z.P_1) \mid \sigma(P_2)])$
$= C[\nu z, sendz, checkz.(sendz\,\mathbf{m} - \mid checkz\,\mathbf{m} - \mid z = \texttt{Channel}\ sendz\ checkz \mid \sigma(P_1)) \mid \sigma(P_2)]$
$\equiv C[\nu z, sendz, checkz.(sendz\,\mathbf{m} - \mid checkz\,\mathbf{m} - \mid z = \texttt{Channel}\ sendz\ checkz \mid \sigma(P_1) \mid \sigma(P_2))]$
$= \sigma_0(\mathbb{D}[\nu z.(P_1 \mid P_2)])$ for some *CHF* $\mathbb{D}$-context C. Thus, $\sigma_0(P) \equiv \sigma_0(P')$.

For rule (nuup2), we have $\sigma_0(\mathbb{D}[\nu z.\nu x.P_1]) = C[\sigma(\nu z.\nu x.P_1)]$
$= C[\nu x, sendx, checkx.(sendx\,\mathbf{m} - \mid checkx\,\mathbf{m} - \mid x = \texttt{Channel}\ sendx\ checkx$
$\qquad \mid \nu z, sendz, checkz.(sendz\,\mathbf{m} - \mid checkz\,\mathbf{m} - \mid z = \texttt{Channel}\ sendz\ checkz \mid \sigma(P_1)))]$
$\equiv C[\nu x, sendx, checkx, \nu z, sendz, checkz.(sendx\,\mathbf{m} - \mid checkx\,\mathbf{m} - \mid x = \texttt{Channel}\ sendx\ checkx$
$\qquad \mid sendz\,\mathbf{m} - \mid checkz\,\mathbf{m} - \mid z = \texttt{Channel}\ sendz\ checkz \mid \sigma(P_1))]$
$\equiv C[\nu z, sendz, checkz.(sendz\,\mathbf{m} - \mid checkz\,\mathbf{m} - \mid x = \texttt{Channel}\ sendz\ checkz$
$\qquad \mid \nu x, sendx, checkx.(sendx\,\mathbf{m} - \mid checkx\,\mathbf{m} - \mid z = \texttt{Channel}\ sendx\ checkx \mid \sigma(P_1)))]$
$\equiv \sigma_0(\mathbb{D}[\nu x.\nu z.P_1])$ for some *CHF* $\mathbb{D}$-context C. Thus, we have $\sigma_0(P) \equiv \sigma_0(P')$.

For rule (commute), we have $\sigma_0(P) \equiv \sigma_0(P')$ since $\sigma_0(\mathbb{D}[P_1 \mid P_2]) = C[(\sigma(P_1) \mid \sigma(P_2))] \equiv C[(\sigma(P_2) \mid \sigma(P_1))] = \sigma_0(\mathbb{D}[P_2 \mid P_1])$ for some $\mathbb{D}$-context C. Thus the claim holds.

For the rule (replunfold), we have:

$$\begin{array}{ll}
\sigma_0(\mathbb{D}[!P]) = & C[\nu f.(\Leftarrow f \mid f = \mathbf{do}\ \{\texttt{future}\ \tau(P); f\})] \ \text{ where } f \notin FV(P) \\
\xrightarrow{sr,cpce} & C[\nu f.(\Leftarrow \mathbf{do}\ \{\texttt{future}\ \tau(P); f\} \mid f = \mathbf{do}\ \{\texttt{future}\ \tau(P); f\})] \\
\xrightarrow{sr,fork}\xrightarrow{sr,beta} & C[\nu f.(\Leftarrow \tau(P) \mid \Leftarrow f \mid f = \mathbf{do}\ \{\texttt{future}\ \tau(P); f\})] \\
\equiv & C[\Leftarrow \tau(P) \mid \nu f.(\Leftarrow f \mid f = \mathbf{do}\ \{\texttt{future}\ \tau(P); f\})] \\
\xrightarrow{sr,*} & C[\sigma(P) \ \mid \nu f.(\Leftarrow f \mid f = \mathbf{do}\ \{\texttt{future}\ \tau(P); f\})] = \sigma_0(\mathbb{D}[P \mid !P])
\end{array}$$

which shows $\sigma_0(\mathbb{D}[!P]) \xrightarrow{sr,*} \sigma_0(\mathbb{D}[P \mid !P])$, using the previous items, and Lemma 6.3 for the last $\xrightarrow{sr,*}$-transformation.

The equivalence $\sigma_0(P) \sim_c \sigma_0(P')$ follows from correctness of the used reduction steps.

Lemma 6.6. *If a closed process P of $\Pi_{\texttt{Stop}}$ is successful, then $\sigma(P) \xrightarrow{sr,*} Q$ where Q is successful. Moreover, $\sigma(P)\downarrow$.*

Proof. Process P must contain *stop* on top-level. Thus the translation generates a thread on top-level that performs `takeMVar` *stop*. Thus $Q \xrightarrow{sr,tmvar} \xrightarrow{sr,pmvar} D[\ \overset{\mathsf{main}}{\Longleftarrow}\ \texttt{return}\ ()]$. For the second part of the lemma, it suffices to observe that σ and τ do not generate a `putMVar` for the MVar *stop* (except in the context C^{σ}_{out}) and thus, the MVar can always be emptied and the main-thread thereafter can perform the `putMVar`-command to become successful.

Proposition 6.7. *Implication of may-convergence: Let P be a closed $\Pi_{\texttt{Stop}}$-process. If $P\downarrow$ then $\tau_0(P)\downarrow$.*

Proof. Lemmas 6.4, 6.5, and 6.6 imply that $\sigma_0(P) \xrightarrow{sr,*} Q$, where Q is successful, using induction on the length of a standard reduction of P, hence $\sigma(P)\downarrow$.

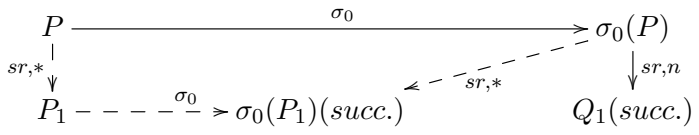
6.3 The Translation Reflects May-Convergence

In this section we show that may-convergence of $\sigma_0(P)$ for a closed $\mathit{\Pi}_{\mathtt{Stop}}$-process P implies may-convergence of P. We show that by rearranging and extending the standard reduction sequence to a successful process, a standard reduction sequence of $\sigma_0(P)$ is obtained that corresponds to a standard reduction sequence of P in the π-calculus. There are three essentially different actions that are executed by the standard reduction sequence of $\sigma_0(P)$, where the single reduction steps may be distributed in the reduction sequence: interaction-reductions, replication of the $!P$-operator, and reducing τ-images to σ-images.

The following properties are easily checked for a translated closed $\mathit{\Pi}_{\mathtt{Stop}}$-process P: $\tau_0(P)$ is closed; $\tau_0(P)$ contains the variable *stop* only in expressions of the form $\Leftarrow \mathtt{takeMVar}\ \mathit{stop}$, and $\tau_0(P)$ is well-formed and well-typed.

Proposition 6.8. *Let $P \in \mathit{\Pi}_{\mathtt{Stop}}$ be closed, $n \in \mathbb{N}$ and $\sigma_0(P) \xrightarrow{sr,n} Q_1$ such that Q_1 is successful. Then there is a standard reduction sequence $P \xrightarrow{sr,*} P_1$ and another standard reduction sequence $\sigma_0(P) \xrightarrow{sr,*} Q_1'$ such that Q_1' and P_1 are successful and $Q_1' = \sigma_0(P_1)$.*

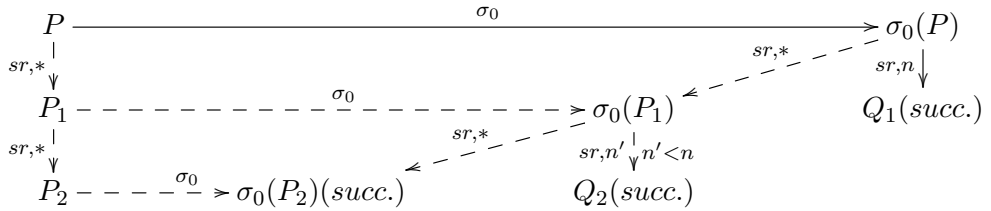
Proof. We show that the following diagram holds by induction on the number of all reduction steps in $\mathit{Red} = \sigma_0(P) \xrightarrow{sr,n} Q_1$.



In this and the diagram below a plain arrow means a given and a dashed arrow means an existing reduction.

The base case is that $\sigma_0(P)$ is of the form $C^{\sigma}_{out}[\mathbb{D}[\Leftarrow \mathtt{takeMVar}\ \mathit{stop}]]$, and it standard-reduces in two steps to a successful process. In this case, the only possible reason is that P contains $\mathtt{Stop}$ in a PCtxt_{π}-context, and σ_0 maps it to this subprocess. Then P is successful.

Now we show the induction step. A picture of the proof structure is:



By the induction hypothesis, we have $P_1{\downarrow}$, but we have to calculate the upper square for all possibilities of actions (interaction reductions, replication, τ-to-σ-reduction) where we rearrange the *CHF*-standard reduction sequence, extend it with missing reduction steps, which do not change success, and construct a corresponding standard reduction in the π-calculus.

Abstractly, a single induction step is intended to do the following: first we identify reduction steps that make a complete or a partial execution that performs one of the three actions. There may be two threads affected by interaction, and otherwise only one thread is affected. We have to identify one particular (distributed) reduction subsequence S that can be executed as a prefix; i.e. Red can be rearranged to $S; \mathit{Red}'$ having the same total effect. There are two possibilities: Either there is such a full subsequence S of reduction steps, or we only identify a prefix of such a subsequence, and the corresponding threads do not have further reduction steps (of the action) until Q is reached. In the first case, everything is fine, and a P_1 can be determined. In the second case, the subsequence has to be extended by the missing reduction steps. Since the thread(s) is/are stopped, these additional reduction steps are independent of all other reduction steps. Hence these can be virtually shifted after Q_1 (perhaps turning into non-sr-reductions) and determine Q_1'. We will add the extra reduction steps to the prefix of the reduction sequence, and then obtain a complete subsequence.

Since the intention of the proof is to construct a reduction sequence between images of translation σ, this enforces that sometimes τ-translated parts have to be sr-reduced to their σ-translation. This is treated in the same way as above.

Referring to Definition 6.1 of translation σ, there are several cases. Let us consider the case that an image of **an ia-reduction** is the first action. Analyzing *Red* shows that, ideally, if we only look for the (tmvar)-, (pmvar)-reductions, the reduction sequence starts as follows

(ia-1) `putMVar (getsend x) y` in thread a;
(ia-2) `takeMVar (getsend x)` in thread b ($\neq a$);
(ia-3) `putMVar (getcheck x) ()` in thread b;
(ia-4) `takeMVar (getcheck x)` in thread a.

However, there may be deviations, like interleaving of reduction steps in parallel threads, or an incomplete subsequence. We show that modifying the reduction sequence *Red* results in another reduction sequence *Red*$'$ that starts with the ideal 4 reductions, also ends in a successful process, such that *Red*$'$ contains less standard-reduction steps.

Now we focus the earliest occurrence in *Red* of an ia-related reduction step. The (ia)-subsequence may be complete, but there may also be incomplete ones. There will be at least one such sub-sequence of reduction steps, which can be executed first, since the first one that makes an MVar-access blocks all others until it is finished. If there is no MVar-access at all (for MVar (`getsend` x)), then we can select an arbitrary thread for extension of the subsequence.

If there is only a proper prefix of the 4 reduction steps in *Red* (in fact these are 8 reduction steps), then we insert the missing MVar-reduction steps into *Red* (and also the necessary other reduction steps), immediately after the subsequence. If we shift the added subsequence after Q_1, we obtain Q_1', which also successful. We put the complete reduction sub-sequence to the front of *Red*, which leads to P_1', and it ends with the successful Q_1'.

Let $\Leftarrow(\tau(P_a))$ and $\Leftarrow(\tau(P_b))$ be the processes that are left by the four reduction steps that simulate the (ia). It remains to check whether $\Leftarrow(\tau(P_a))$ and $\Leftarrow(\tau(P_b))$ reduce to $\sigma(P_a)$ and $\sigma(P_b)$, respectively. Here we use the same construction as above: The reduction steps that are already in the sequence are shifted to the left. Since the threads are blocked if there are missing reductions, we can add the missing reduction steps to the reduction sequence, keeping the property that the resulting process of the whole sequence is successful. For details, we refer to Lemma 6.3. We obtain the upper square of the diagram, where also the number of reductions is strictly smaller for the remaining reduction sequence.

We now consider the case that the reduction is an image of a (**replunfold**) step. Then the corresponding reduction is the (fork)-reduction. In fact, it is a sequence $\xrightarrow{cpce} \cdot \xrightarrow{fork} \cdot \xrightarrow{beta}$. We can, following the reduction pattern in the proof of Lemma 6.5, move the reduction steps to the front, which are corresponding to a single replication. If reduction steps are missing, we can add the missing steps without disturbing the final success. Also, we can add the necessary reductions that may be missing to turn the $\tau(P_r)$ into $\sigma(P_r)$. Again we can construct the square diagram, as requested, where also in this case, the process that represents the final success may have changed.

Finally, we can apply the induction hypothesis, since the combined measure is strictly reduced, and we thus obtain a standard reduction in $\Pi_{\texttt{Stop}}$ to a successful process. □

Propositions 6.7 and 6.8 imply:

Proposition 6.9. *Let $P \in \Pi_{\texttt{Stop}}$ be closed. Then τ_0 is convergence-equivalent for $\downarrow$, i.e. $P\downarrow$ is equivalent to $\tau_0(P)\downarrow$. This also implies that $P\Uparrow$ is equivalent to $\tau(P)\Uparrow$.*

6.4 Equivalence of Should-Convergence of the Translation

We argue that the translation τ leaves should-convergence invariant, where we work with may-divergence and where Proposition 6.9 is very useful since it is the induction base.

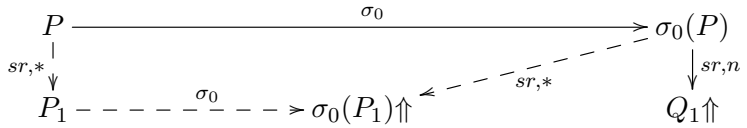
Proposition 6.10. *Let $P \in \Pi_{\mathtt{Stop}}$ be closed, $n \in \mathbb{N}$ and $P \xrightarrow{sr,n} P_1$ such that $P_1\Uparrow$. Then there is a standard reduction sequence $\sigma(P) \xrightarrow{sr,*} \sigma(P_1)$ with $\sigma(P_1)\Uparrow$*

Proof. The proof is similar to the implication proof for may-convergence (Proposition 6.7) with the difference that the reduction sequence ends in a must-divergent process. Proposition 6.9 shows the base case. The induction step is almost the same as in the proof of Proposition 6.7, with the difference that the reduction sequences end with must-divergent processes. □

The last case is to show that $\sigma(P)\uparrow \implies P\uparrow$. This is almost similar to the arguments for $\sigma(P)\downarrow \implies P\downarrow$ where some more arguments are needed to show that the final process remains must-divergent after the rearrangements and additions to the reduction sequence.

Proposition 6.11. *Let $P \in \Pi_{\mathtt{Stop}}$ be closed, $n \in \mathbb{N}$ and $\sigma_0(P) \xrightarrow{sr,n} Q_1$ such that $Q_1\Uparrow$. Then there is a standard reduction sequence $P \xrightarrow{sr,*} P_1$ and another standard reduction sequence $\sigma(P) \xrightarrow{sr,n} Q_1'$ such that $Q_1'\Uparrow$ and $Q_1' = \sigma_0(P_1')$.*

Proof. It is sufficient to show the following diagram by induction on n.



An additional argument is that $Q\Uparrow$ and $Q \xrightarrow{*} Q'$ implies $Q'\Uparrow$, which is required for the rearrangement and extension of the standard reduction sequence. Using these arguments the proof is otherwise completely analogous to the proof of Proposition 6.8. □

Proposition 6.12. *Let $P \in \Pi_{\mathtt{Stop}}$ be closed. Then τ_0 is convergence-equivalent for $\Downarrow$, i.e. $P\Downarrow$ is equivalent to $\tau_0(P)\Downarrow$.*

7 Conclusion

We have shown that there is a translation from the π-calculus into *CHF*, which is an embedding for closed processes w.r.t. $\sim_c$, and preserves may-convergence behavior as well as the should-convergence behavior. The translation is rather strong, since even for open processes, it is adequate w.r.t. the respective contextual equivalences.

For further work, we may consider extended variants of the π-calculus. We are convinced that adding recursion and sums can easily be built into the translation, while it might be challenging to encode (name) matching operators.

References

AG97. Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The Spi calculus. In Richard Graveman, Philippe A. Janson, Clifford Neumann, and Li Gong, editors, *CCS '97*, pages 36–47. ACM, 1997.

BBP95. Richard Banach, J. Balázs, and George A. Papadopoulos. A translation of the pi-calculus into MONSTR. *J.UCS*, 1(6):339–398, 1995.

com19. Haskell community. Haskell main website, 2019. `www.haskell.org`.

FG02. C. Fournet and G. Gonthier. The join calculus: A language for distributed mobile programming. In *APPSEM'00*, volume 2395 of *LNCS*, pages 268–332. Springer, 2002.

Han18. Axel Hanczak. Untersuchung einer Übersetzung des Pi-Kalküls in die nebenläufige funktionale Programmiersprache Concurrent Haskell with Futures, *english:* investigating a translation of the pi-calcuus into the concurrent functional programming language Concurrent Haskell, 2018. Goethe-University Frankfurt, Germany, Dept. Computer Science, B.Sc. thesis, in German.

Lan96. C. Laneve. On testing equivalence: May and must testing in the join-calculus. Technical Report UBLCS 96-04, University of Bologna, 1996.

Mil99. Robin Milner. *Communicating and mobile systems - the Pi-calculus.* Cambridge University Press, 1999.

MPW92. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I & II. *Inform. and Comput.*, 100(1):1–77, 1992.

PGF96. Simon L. Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Proc. 23rd ACM POPL 1996*, pages 295–308. ACM, 1996.

Pri95. Corrado Priami. Stochastic pi-calculus. *Comput. J.*, 38(7):578–589, 1995.

Sab14. David Sabel. Structural Rewriting in the pi-Calculus. In *First International Workshop on Rewriting Techniques for Program Transformations and Evaluation*, volume 40 of *OpenAccess Series in Informatics (OASIcs)*, pages 51–62, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

SS15. David Sabel and Manfred Schmidt-Schauß. Observing success in the pi-calculus. In *2nd WPTE 2015*, volume 46 of *OASICS*, pages 31–46, 2015.

SSNS15. Manfred Schmidt-Schauß, David Sabel, Joachim Niehren, and Jan Schwinghammer. Observational program calculi and the correctness of translations. *Theor. Comput. Sci.*, 577:98–124, 2015.

SSNSS08. Manfred Schmidt-Schauß, Joachim Niehren, Jan Schwinghammer, and David Sabel. Adequacy of compositional translations for observational semantics. In *Proc. IFIP TCS'08*, volume 273 of *IFIP*, pages 521–535. Springer, 2008.

SSS11. David Sabel and Manfred Schmidt-Schauß. A contextual semantics for Concurrent Haskell with futures. In *Proc. PPDP'11*, pages 101–112. ACM, 2011.

SSS12. David Sabel and Manfred Schmidt-Schauß. Conservative concurrency in Haskell. In *Proc. 27th IEEE LICS 2012*, pages 561–570. IEEE, 2012.

SW01a. Davide Sangiorgi and David Walker. On barbed equivalences in pi-calculus. In *Proc. CONCUR 2001*, volume 2154 of *LNCS*, pages 292–304. Springer, 2001.

SW01b. Davide Sangiorgi and David Walker. *The π-calculus: a theory of mobile processes.* Cambridge university press, 2001.

YRS04. Ping Yang, C. R. Ramakrishnan, and Scott A. Smolka. A logical encoding of the pi-calculus: model checking mobile processes using tabled resolution. *STTT*, 6(1):38–66, 2004.

A Barbed Convergence Testing and Equivalence

In [SS15], the following Context Lemma was proved, where σ are name-to-name substitutions:

Theorem A.1 ([SS15]). *For all processes $P, Q \in \Pi_{\texttt{Stop}}$:*

- *If for all σ, R: $\sigma(P) \mid R \leq_{\downarrow} \sigma(Q) \mid R$, then $P \leq_{c,\downarrow} Q$.*
- *If for all σ, R: $\sigma(P) \mid R \leq_{\downarrow} \sigma(Q) \mid R \wedge \sigma(P) \mid R \leq_{\Downarrow} \sigma(Q) \mid R$, then $P \leq_c Q$.*

Example A.2. We show the equivalence $x(y).\texttt{Stop} \mid \bar{x}(z).\texttt{Stop} \sim_c \texttt{Stop}$. Clearly for all σ and all processes R we have $\sigma(\texttt{Stop}) \mid R$ is successful and thus $\sigma(\texttt{Stop}) \mid R\Downarrow$ and $\sigma(\texttt{Stop}) \mid R\downarrow$. We also have that $\sigma(x(y).\texttt{Stop} \mid \bar{x}(z).\texttt{Stop}) \mid R\downarrow$, since the process reduces in one step to $\texttt{Stop} \mid R$. We finally, observe that it is impossible to reduce $\sigma(x(y).\texttt{Stop} \mid \bar{x}(z).\texttt{Stop}) \mid R$ into a must-divergent process, since the process becomes successful if one of the components $\sigma(x(y).\texttt{Stop})$ or $\sigma(\bar{x}(y).\texttt{Stop})$ is part of the redex, and otherwise (the interaction between these two processes is always possible). Thus, $\sigma(x(y).\texttt{Stop} \mid \bar{x}(z).\texttt{Stop}) \mid R$ is must-convergent. Hence all preconditions of the context lemma (Theorem A.1) hold and the equivalence holds.

For stop-free processes this notion of contextual equivalence coincides with so-called barbed testing equivalence, which observes whether a process may- or should-reduces to a process that has a free input on a fixed channel name (see [SS15]):

Definition A.3. *Let Π be the subcalculus of $\Pi_{\texttt{Stop}}$ that does not have the constant* Stop *as a syntactic construct. Processes, contexts, reduction, structural congruences are accordingly adapted for Π.*

Let $P \in \Pi$ and $x \in \mathcal{N}$. A process P has a barb on input *x (written as $P{\restriction^x}$) iff $P \equiv \nu\mathcal{X}.(x(y).P' \mid P'')$ where $x \notin \mathcal{X}$. We write $P{\downarrow_x}$ iff there exists P' such that $P \xrightarrow{sr,*} P'$ and $P'{\restriction^x}$. We write $P{\Downarrow_x}$ iff for all P' with $P \xrightarrow{sr,*} P'$ also $P'{\downarrow_x}$ holds. We write $P{\Uparrow_x}$ iff $P{\downarrow_x}$ does not hold, and we write $P{\uparrow_x}$ iff $P{\Downarrow_x}$ does not hold.*

For a name $x \in \mathcal{N}$, barbed may- and should-testing preorder *$\leq_{c,barb}$ and* barbed may- and should-testing equivalence *$\sim_{c,barb}$ are defined as $\leq_{c,barb} := \leq_{c,\downarrow_x} \cap \leq_{c,\Downarrow_x}$ and $\sim_{c,barb} := \leq_{c,barb} \cap (\leq_{c,barb})^{-1}$ where for $\xi \in \{\downarrow_x, \Downarrow_x, \uparrow_x, \Uparrow_x\}$ and $P, Q \in \Pi$ the inequality $P \leq_{c,\xi} Q$ holds iff for all contexts $C \in \Pi : C[P]\xi \implies C[Q]\xi$.*

Theorem A.4 ([SS15]). *For all processes $P, Q \in \Pi$: $P \leq_{c,barb} Q \iff P \leq_c Q$, and hence also $P \sim_{c,barb} Q \iff P \sim_c Q$.*

B Proof of Lemma 6.3

Lemma B.1 (This is Lemma 6.3). *For all $\Pi_{\texttt{Stop}}$-processes P: 1. $\tau_0(P) \xrightarrow{sr,*} \sigma_0(P)$ and 2. $\Leftarrow \tau(P) \xrightarrow{sr,*} \sigma(P)$.*

Proof. For the first part, it suffices to verify that $\tau_0(P)$ is a process that reduces as follows:

$$
\begin{array}{ll}
 & \tau_0(P) = z \overset{\mathsf{main}}{\Longleftarrow} \textbf{do}\ \{\mathit{stop} \leftarrow \texttt{newMVar}\ ();\texttt{future}\ \tau(P);\texttt{putMVar}\ \mathit{stop}\ ()\} \\
\xrightarrow{sr,nmvar}\xrightarrow{sr,beta} & \nu \mathit{stop}.z \overset{\mathsf{main}}{\Longleftarrow} \textbf{do}\ \{\texttt{future}\ \tau(P);\texttt{putMVar}\ \mathit{stop}\ ()\} \mid \mathit{stop}\,\mathbf{m}\,() \\
\xrightarrow{sr,fork}\xrightarrow{sr,beta} & \nu \mathit{stop}.z \overset{\mathsf{main}}{\Longleftarrow} \texttt{putMVar}\ \mathit{stop}\ () \mid \mathit{stop}\,\mathbf{m}\,() \mid \Leftarrow \tau(P) \\
\xrightarrow{sr,*} & \nu \mathit{stop}.z \overset{\mathsf{main}}{\Longleftarrow} \texttt{putMVar}\ \mathit{stop}\ () \mid \mathit{stop}\,\mathbf{m}\,() \mid \sigma(P) = \sigma_0(P)
\end{array}
$$

The $\xrightarrow{sr,*}$-sequence is derived from the second part of the lemma, and by plugging-in the reduction sequence in the larger context.

It remains to show the second part. We show this by induction on the size of P and by checking all the cases.

- If P starts with an input or an output prefix or is `Stop`, then $\Leftarrow\tau(P) = \sigma(P)$, and thus the claim holds.
- If P is the silent process, then $\Leftarrow\tau(P) = \Leftarrow \texttt{return}\ () \xrightarrow{sr,unIO} \nu x.x = x() = \sigma(P)$
- If P is a parallel composition $P_1 \mid P_2$, then

$$\begin{array}{l} \Leftarrow\tau(P_1 \mid P_2) = \Leftarrow \mathbf{do}\ \{\texttt{future}\ \tau(P_2); \tau(P_1)\} \\ \xrightarrow{sr,fork}\xrightarrow{sr,beta}\ \Leftarrow\tau(P_2) \mid \Leftarrow\tau(P_1) \xrightarrow{sr,*} \sigma(P_2) \mid \sigma(P_1) = \sigma(P_1 \mid P_2) \end{array}$$

  The final $\xrightarrow{sr,*}$ sequence is obtained as follows: by the induction hypothesis we have $\Leftarrow\tau(P_i) \xrightarrow{sr,*} \sigma(P_i)$ for $i = 1, 2$ and we combine these sr-reductions (by processing them sequentially) and get $\Leftarrow\tau(P_2) \mid \Leftarrow\tau(P_1) \xrightarrow{sr,*} \sigma(P_2) \mid \Leftarrow\tau(P_1) \xrightarrow{sr,*} \sigma(P_2) \mid \sigma(P_1)$
- If the process starts with νx, then

$$\begin{array}{l} \Leftarrow\tau(\nu x.P) = \Leftarrow \mathbf{do}\ \{sendx \leftarrow \texttt{newMVar}\ \bot; checkx \leftarrow \texttt{newMVar}\ (); \texttt{takeMVar}\ sendx; \\ \qquad\qquad \texttt{takeMVar}\ checkx; \texttt{letrec}\ x = \texttt{Channel}\ sendx\ checkx\ \texttt{in}\ \tau(P)\} \\ \xrightarrow{sr,nmvar}\xrightarrow{sr,beta}\xrightarrow{sr,nmvar}\xrightarrow{sr,beta} \Leftarrow \mathbf{do}\ \{\texttt{takeMVar}\ sendx; \texttt{takeMVar}\ checkx; \\ \qquad\qquad\qquad\qquad \texttt{letrec}\ x = \texttt{Channel}\ sendx\ checkx\ \texttt{in}\ \tau(P)\} \\ \qquad\qquad\qquad\qquad \mid checkx\ \mathbf{m}\ () \mid sendx\ \mathbf{m}\ () \\ \xrightarrow{sr,tmvar}\xrightarrow{sr,beta}\xrightarrow{sr,tmvar}\xrightarrow{sr,beta} \Leftarrow \texttt{letrec}\ x = \texttt{Channel}\ sendx\ checkx\ \texttt{in}\ \tau(P) \\ \qquad\qquad\qquad\qquad \mid checkx\ \mathbf{m} - \mid sendx\ \mathbf{m} - \\ \xrightarrow{sr,mkbinds} \Leftarrow\tau(P) \mid x = \texttt{Channel}\ sendx\ checkx\ \mid checkx\ \mathbf{m} - \mid sendx\ \mathbf{m} - \\ \xrightarrow{sr,*} \sigma(P) \mid x = \texttt{Channel}\ sendx\ checkx\ \mid checkx\ \mathbf{m} - \mid sendx\ \mathbf{m} - \end{array}$$

  Note that the (tmvar)-reductions are deterministic (i.e. (dtmvar)-transformations), since there is no alternative, since the names of the visibility of the MVars and the potential accesses leave only one possibility. The final standard reduction sequence is obtained by first applying the induction hypothesis for P to derive $\Leftarrow\tau(P) \xrightarrow{sr,*} \sigma(P)$ and then observing that the same reduction sequence can be performed if there are more parallel components.
- If the process is a replication, then

$$\begin{array}{l} \Leftarrow\tau(!P) = \Leftarrow \texttt{letrec}\ f = \mathbf{do}\ \{\texttt{future}\ \tau(P); f\}\ \texttt{in}\ f \\ \xrightarrow{sr,mkbinds} \nu f. \Leftarrow f \mid f = \mathbf{do}\ \{\texttt{future}\ \tau(P); f\} \quad = \quad \sigma(!P). \end{array}$$

This finishes the induction proof.

Lemma 6.3 and correctness of (fork), (nmvar), (dtmvar), (beta), (cpce), and (mkbinds) imply:

Proposition B.2. *For all $\Pi_{\texttt{Stop}}$-processes P: 1. $\sigma(P) \sim_c \Leftarrow\tau(P)$ and 2. $\sigma_0(P) \sim_c \tau_0(P)$.*

C Reduction Sequence for Lemma 6.4

We show that $\sigma_0(P) \xrightarrow{sr,16} Q$ in Lemma 6.4:

$$\begin{array}{l} \sigma_0(P) = C^{\sigma}_{out}[D_1[\nu x, sendx, checkx. sendx\ \mathbf{m} - \mid checkx\ \mathbf{m} - \mid x{=}\texttt{Channel}\ sendx\ checkx \\ \qquad \mid D_2[\Leftarrow \mathbf{do}\ \{\texttt{putMVar}\ (\texttt{getsend}\ x)\ z; \texttt{takeMVar}\ (\texttt{getcheck}\ x); \tau(P_s)\} \\ \qquad\quad \mid \Leftarrow \mathbf{do}\ \{y{\leftarrow}\texttt{takeMVar}\ (\texttt{getsend}\ x); \texttt{putMVar}\ (\texttt{getcheck}\ x)\ (); \tau(P_r)\}]]] \end{array}$$

$$\begin{array}{l} \xrightarrow{sr,cpce}\xrightarrow{sr,case} C^{\sigma}_{out}[D_1[\nu x, sendx, checkx. sendx\ \mathbf{m} - \mid checkx\ \mathbf{m} - \mid x{=}\texttt{Channel}\ sendx\ checkx \\ \qquad \mid D_2[\Leftarrow \mathbf{do}\ \{\texttt{putMVar}\ sendx\ z; \texttt{takeMVar}\ (\texttt{getcheck}\ x); \tau(P_s)\} \\ \qquad\quad \mid \Leftarrow \mathbf{do}\ \{y{\leftarrow}\texttt{takeMVar}\ (\texttt{getsend}\ x); \texttt{putMVar}\ (\texttt{getcheck}\ x)\ (); \tau(P_r)\}]]] \end{array}$$

$$\xrightarrow{sr,cpce}\xrightarrow{sr,case} C^{\sigma}_{out}[D_1[\nu x, sendx, checkx.sendx\,\mathbf{m} - \mid checkx\,\mathbf{m} - \mid x{=}\texttt{Channel}\ sendx\ checkx$$
$$\mid D_2[\Leftarrow \mathbf{do}\ \{\texttt{putMVar}\ sendx\ z; \texttt{takeMVar}\ (\texttt{getcheck}\ x); \tau(P_s)\}$$
$$\mid \Leftarrow \mathbf{do}\ \{y \leftarrow \texttt{takeMVar}\ (sendx); \texttt{putMVar}\ (\texttt{getcheck}\ x)\ (); \tau(P_r)\}]]]$$

$$\xrightarrow{sr,pmvar}\xrightarrow{sr,beta} C^{\sigma}_{out}[D_1[\nu x, sendx, checkx.sendx\,\mathbf{m}\,z \mid checkx\,\mathbf{m} - \mid x{=}\texttt{Channel}\ sendx\ checkx$$
$$\mid D_2[\Leftarrow \mathbf{do}\ \{\texttt{takeMVar}\ (\texttt{getcheck}\ x); \tau(P_s)\}$$
$$\mid \Leftarrow \mathbf{do}\ \{y \leftarrow \texttt{takeMVar}\ (sendx); \texttt{putMVar}\ (\texttt{getcheck}\ x)\ (); \tau(P_r)\}]]]$$

$$\xrightarrow{sr,tmvar}\xrightarrow{sr,beta} C^{\sigma}_{out}[D_1[\nu x, sendx, checkx.sendx\,\mathbf{m} - \mid checkx\,\mathbf{m} - \mid x{=}\texttt{Channel}\ sendx\ checkx$$
$$\mid D_2[\Leftarrow \mathbf{do}\ \{\texttt{takeMVar}\ (\texttt{getcheck}\ x); \tau(P_s)\}$$
$$\mid \Leftarrow (\mathbf{do}\ \{\texttt{putMVar}\ (\texttt{getcheck}\ x)\ (); \tau(P_r)\})[z/y]]]]$$

$$\xrightarrow{sr,cpce}\xrightarrow{sr,case} C^{\sigma}_{out}[D_1[\nu x, sendx, checkx.sendx\,\mathbf{m} - \mid checkx\,\mathbf{m} - \mid x{=}\texttt{Channel}\ sendx\ checkx$$
$$\mid D_2[\Leftarrow \mathbf{do}\ \{\texttt{takeMVar}\ checkx; \tau(P_s)\}$$
$$\mid \Leftarrow (\mathbf{do}\ \{\texttt{putMVar}\ (\texttt{getcheck}\ x)\ (); \tau(P_r)\})[z/y]]]]$$

$$\xrightarrow{sr,cpce}\xrightarrow{sr,case} C^{\sigma}_{out}[D_1[\nu x, sendx, checkx.sendx\,\mathbf{m} - \mid checkx\,\mathbf{m} - \mid x{=}\texttt{Channel}\ sendx\ checkx$$
$$\mid D_2[\Leftarrow \mathbf{do}\ \{\texttt{takeMVar}\ checkx; \tau(P_s)\}$$
$$\mid \Leftarrow (\mathbf{do}\ \{\texttt{putMVar}\ (checkx)\ (); \tau(P_r)\})[z/y]]]]$$

$$\xrightarrow{sr,pmvar}\xrightarrow{sr,beta} C^{\sigma}_{out}[D_1[\nu x, sendx, checkx.sendx\,\mathbf{m} - \mid checkx\,\mathbf{m}\,() \mid x{=}\texttt{Channel}\ sendx\ checkx$$
$$\mid D_2[\Leftarrow \mathbf{do}\ \{\texttt{takeMVar}\ checkx; \tau(P_s)\} \mid \Leftarrow \tau(P_r)[z/y]]]]$$

$$\xrightarrow{sr,tmvar}\xrightarrow{sr,beta} C^{\sigma}_{out}[D_1[\nu x, sendx, checkx.sendx\,\mathbf{m} - \mid checkx\,\mathbf{m} - \mid x{=}\texttt{Channel}\ sendx\ checkx$$
$$\mid D_2[\Leftarrow \tau(P_s) \mid \Leftarrow \tau(P_r)[z/y]]]]$$
$$= Q$$