Embedding the Pi-Calculus into a Concurrent Functional Programming Language [⋆]

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

[1] Goethe-University Frankfurt, Germany
{schauss,sabel}@ki.cs.uni-frankfurt.de
[2] LMU Munich, Germany david.sabel@lmu.de

Technical Report Frank-60

Research group for Artificial Intelligence and Softwa3re Technology
Institut für Informatik,
Fachbereich Informatik und Mathematik,
Johann Wolfgang Goethe-Universität,
Postfach 11 19 32, D-60054 Frankfurt, Germany

July 13, 2020
This version corrects and extends the report from May 2020.

Abstract. Correctness of program transformations and translations in concurrent programming is the focus of our research. In this case study the relation of the synchronous pi-calculus and a core language of Concurrent Haskell (CH) with asynchronous communication is investigated. We show that CH embraces the synchronous pi-calculus. The formal foundations are contextual semantics in both languages, where may- as well as should-convergence are observed. We succeed in defining and proving smart properties of a particular translation mapping the synchronous pi-calculus into CH. This implies that pi-processes are error-free if and only if their translation is an error-free CH-program Our result shows that the chosen semantics is not only powerful, but can also be applied in concrete and technically complex situations. The developed translation uses private names. We also automatically check potentially correct translations that use global names instead of private names. As a complexity parameter we use the number of MVars introduced by the transformation, where MVars are synchronized 1-place buffers. The automated refutation of incorrect translations leads to a classification of potentially correct translations, and to the conjecture that one global MVar is insufficient.

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

[⋆] The first author is supported by the Deutsche Forschungsgemeinschaft (DFG) under grant SCHM 986/11-1.The second author is supported by the Deutsche Forschungsgemeinschaft (DFG) under grant SA2908/3-1.

1 Introduction

Motivation and Goals. Our goals are the correct compilation and optimization of programs, correctness of transformations, the comparison of programming languages, and properties of translations. We already showed that contextual semantics of concurrent programming languages is powerful and that it is effectively possible to verify correctness of transformations [15,25,26], also under the premise not to worsen the runtime [31]. Testing may- and should-convergence in the contextual semantics turns out to be rather powerful, and has been used for concurrent functional calculi to transform and optimize programs, where for a large set of program transformations the computations are feasible.

We want to apply our techniques to the translation of programming languages, which is even more ambitious, since the contextual semantics is defined independently for the involved languages and their respective evaluations. In [30,32] we developed notions of correctness of translations w.r.t. contextual semantics and gave several (introductory) examples. In [33] we applied these techniques in the context of concurrency, but however, there the source and the target language are quite similar (they are variations of the same concurrent call-by-value lambda calculus with shared memory). To underpin the techniques and to take a next step in our research we were looking for more interesting examples where source and target language are both concurrent but also quite different. That is why we translate a synchronous message passing model into an (asynchronous) shared memory model. Thus, as a case study we investigate the potential translations of a synchronous π-calculus into a core-language of Concurrent Haskell, the calculus CH.

The contextual semantics of concurrent programming languages is a generalization of the extensionality principle of functions. The test for a program P is whether $C[P]$ for all program-contexts C successfully terminates (converges) or not, which usually means that the standard reduction sequence ends with a value. For a concurrent program P, we use two observations: *may-convergence* ($P{\downarrow}$) – at least one execution path terminates successfully, and *should-convergence*[3] ($P{\Downarrow}$) – every intermediate state of a reduction sequence may-converges. For two processes P_1, P_2, $P_1 \leq_c P_2$ holds iff for all contexts $C[\cdot]$: $(C[P_1]{\downarrow} \implies C[P_2]{\downarrow})$ and $(C[P_1]{\Downarrow} \implies C[P_2]{\Downarrow})$, and $P_1 \sim_c P_2$ (are equivalent) iff $P_1 \leq_c P_2$ and $P_2 \leq_c P_1$. Processes P_1 and P_2 are contextually equivalent, $P_1 \sim_c P_2$, iff $P_1 \leq_c P_2$ and $P_2 \leq_c P_1$. Showing equal expressivity of two (concurrent) calculi by a translation τ is founded on properties of τ: this requires that may- and should-convergence make sense in each calculus. An important property is adequacy (see Definition 4.4), which holds if $\tau(P_1) \leq_c \tau(P_2)$ implies $P_1 \leq_c P_2$, for the contextual preorders and for all processes P_1, P_2. Full-abstractness, i.e. $\tau(P_1) \leq_c \tau(P_2)$ iff $P_1 \leq_c P_2$, only holds in the rare case that the two calculi are more or less the

[3] An alternative observation is must-convergence (all execution paths terminate). The advantages of equivalence notions based on may- and should-convergence are invariance under fairness restrictions, preservation of deadlock-freedom, and equivalence of busy-wait and wait-until behavior (see e.g. [33]).

same. Translating the π-calculus into *CH* requires to find at least one very good translation. Our translation τ_0 (defined and analyzed in Sect. 4) is one of these.

Source and Target Calculi. The well-known π-calculus [14,13,29] 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, and processes and links between processes can be dynamically created and removed 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 various applications, like reasoning about cryptographic protocols [1], applications in molecular biology [20], and distributed computing [11,7]. The π-calculus also permits the study of intrinsic principles and semantics of concurrency, of concurrent programming and the inherent nondeterministic behavior of mobile sand 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. To observe termination of a process, the investigated π-calculus has a constant `Stop` which signals successful termination.

One important reason, why use Concurrent Haskell as a target language, and not sequential Haskell, is that for a correct translation the target language must be able to perform a parallel convergence test [21,19], since the π-calculus has one (i.e. by the context $[\cdot] \mid [\cdot]$. Core-Haskell with erratic choice (i.e., a construct that permits to choose between two expressions) appears to be more expressive, however, this and similar calculi do *not* have a parallel-convergence testing context $P[\cdot,\cdot]$. Informally, the reason is that if $P[v,\bot]$ as well as $P[\bot,v]$ converge for values v, then in such programming languages $P[e_1,e_2]$ has to converge independently of expressions e_1, e_2 (in particular, $P[\bot,\bot]$ must also converge). Note that there are stronger nondeterministic primitives, like McCarthy's `amb`-operator [12], which can express parallel-convergence testing contexts (see e.g. [24]). Concurrent Haskell [18,9] is very powerful and has a parallel-convergence testing context, which also holds for *CH*. The calculus *CH*, is a process calculus where threads evaluate expressions from a lambda calculus extended by data constructors, case-expressions, recursive let-expressions, and Haskell's seq-operator. Also monadic operations (sequencing and creating threads) are available. The shared memory is modeled by MVars (mutable variables) which are one-place buffers that can be either filled or empty and are blocking if a thread tries to fill a full buffer or to empty an empty buffer. The calculus *CH* is a variant (or a subcalculus) of the calculus *CHF* [25,26] which extends Concurrent Haskell by futures. A major advantage of this approach is that we can reuse studies and results on the contextual semantics of *CHF* also for *CH*.

Details and Variations of the Translation One main issue for a correct translation from π-processes to *CH*-programs is to correctly encode the synchronous communication of the π-calculus. The problem is that the so-called MVars in *CH* (see also [18]) have an asynchronous behavior. To implement synchronous communication the weaker synchronization property of MVars has to be exploited, where we must be aware of the potential interferences of the executions of other translated communications on the same channel. The task of finding such translations is reminiscent of the channel-encoding used in [18], but, however, there an asynchronous channel is implemented while we look for synchronous communication.

We provide a translation τ_0 which uses a private MVar per channel to ensure that no other sender or receiver can interfere with the interaction. A similar idea was used in [10,3] in a different scenario. We succeed in mathematically confirming that the translation τ_0 has strong and nice semantic properties.

Since we are also interested in simpler translations (e.g. using private names is complex, since it generates a new MVar for each communication), we looked for correct translations with a fixed and static number of MVars per channel in the π-calculus. Since this task is too complex and error-prone for hand-crafting, we automated it by implementing a procedure to rule out incorrect translations[4]. Thereby we assumed per communication channel a single MVar for exchanging the channel-name and perhaps several additional MVars of unit type to perform checks whether the message was send / or received (we call them check-MVars).

The outcomes of our automated search are a correct translation that uses two check-MVars, where one is used as a mutex between all senders or receivers on one channel, and correct translations using three additional MVars where the filling and emptying operations for each MVar must not come from the same sender or receiver. The experiments lead to the conjecture that there is no translation using only one check-MVar.

Results. Our novel results are a translation τ_0 from the π-calculus into CH, adequacy of the open translation τ (Theorem 4.10), and full abstraction of τ on closed π-processes (Theorem 4.12). 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 (Definition 4.4) for open processes is also novel extending our work in [30,32]. We further define a quite general formalism for the representation of translations with global names and analyze different classes of such translations by an automated tool. In particular, we show correctness of two particular of these translations in Theorems 5.9 and 5.11.

The discovered correct translations look quite simple and their correctness seems to be quite intuitive. However, our experience is that searching for correct translations is quite hard. We “found” several translations which also were simple and seemed to be intuitively correct. However, they were wrong. Our automated tool helped us to rule out wrong translations.

Further Related Work. Encodings of synchronous communication by asynchronous communication using a private name mechanism are given in [10,3] for (variants of the) π-calculus. Our idea of the translation τ_0 similarly uses a private MVar to encode the channel based communication, but clearly our setting is different, since our target language is Concurrent Haskell. Encodings between

[4] The tool and some output generated by the tool are available via https://gitlab.com/davidsabel/refute-pi.

π-calculi with synchronous and with asynchronous communication were, for instance, already considered in [10,3,17] where encodability results are obtained for the π-calculus without sums [10,3], while in [17] the expressiveness of synchronous and asynchronous communication in the π-calculus with *mixed sums* was compared and non-encodability is a main result. Work on translations of the π-calculus into programming calculi and logical systems is [2], where a translation into a graph-rewriting calculus is given and soundness and completeness w.r.t. the operational behavior is proved. The article [34] shows a translation and a proof that the π-calculus is exactly operationally represented. There are several works on session types which are related to the π-calculus, e.g., [16] studies encodings from a session calculus into PCF extended by concurrency and effects and also an embedding in the other direction, mapping PCF extended by effects into a session calculus. The result is a (strong) operational correspondence between both calculi. Encodings of CML-events in Concurrent Haskell using MVars are published in [22,5]. This approach is more high-level than ours (since we are working directly with the π-calculus). In [5] correctness of a distributed protocol for selective-communication w.r.t. an excerpt of CML is shown, and correctness of implementing the protocol in the π-calculus is proved. The protocol is implemented in Concurrent-Haskell, but correctness of this part is not proven, since the paper focuses to show that CML-events are implementable in languages with first-order message-passing which is different from our focus (translating the π-calculus into *CH*).

Outline. We introduce both calculi in Sects. 2 and 3 and the translation using private names in Sect. 4. In Sect. 5 we treat translations with global names and present our automated tool. We conclude and discuss future work in Sect. 6.

2 The π-Calculus with Stop

We explain the synchronous π-calculus [14,13,29] without sums and with replication in a variant extended with a constant `Stop` [27], that signals successful termination of the whole π-calculus program. The π-calculus without `Stop` and with so-called barbed convergences [28] are equivalent w.r.t. contextual semantics (see Theorem D.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`.

Definition 2.1. *Let $\mathcal{N}$ be a countable set of (channel) names. The syntax of processes is shown in Fig. 1. Free names $FN(P)$, bound names $BN(P)$, and α-equivalence $=_\alpha$ in $\Pi_{\texttt{Stop}}$ are as usual in the π-calculus. A process P is closed if $FN(P) = \emptyset$. 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 $\overline{x}y.P$ 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, and after

$$\begin{aligned}
P, Q \in \Pi_{\texttt{Stop}} &::= \nu x.P \mid \overline{x}y.P \mid x(y).P \mid\, !P \mid P \boldsymbol{|} Q \mid 0 \mid \texttt{Stop}\\
C \in \Pi_{\texttt{Stop},C} &::= [\cdot] \mid \bar{x}(y).C \mid x(y).C \mid C \boldsymbol{|} P \mid P \boldsymbol{|} C \mid !C \mid \nu x.C\\
\mathbb{D} \in \mathit{PCtxt}_\pi &::= [\cdot] \mid \mathbb{D} \boldsymbol{|} P \mid P \boldsymbol{|} \mathbb{D} \mid \nu x.\mathbb{D}.
\end{aligned}$$

Fig. 1. Syntax of processes $\Pi_{\texttt{Stop}}$, process contexts $\Pi_{\texttt{Stop},C}$ and reduction contexts PCtxt_π where x, y are names.

Interaction rule:
(ia) $x(y).P \boldsymbol{|} \overline{x}z.Q \xrightarrow{ia} P[z/y] \boldsymbol{|} Q$

Closure: If $P \equiv \mathbb{D}[P'], P' \xrightarrow{ia} Q', \mathbb{D}[Q'] \equiv Q,$ and $\mathbb{D} \in \mathit{PCtxt}$ then $P \xrightarrow{sr} Q$

Fig. 2. Reduction rule and standard reduction in $\Pi_{\texttt{Stop}}$

$$\begin{aligned}
P &\equiv Q, \text{ if } P{=_\alpha}Q\\
P \boldsymbol{|} (Q \boldsymbol{|} R) &\equiv (P \boldsymbol{|} Q) \boldsymbol{|} R\\
\nu x.(P \boldsymbol{|} Q) &\equiv P \boldsymbol{|} \nu x.Q,\\
&\quad\text{if } x \notin \mathit{FN}(P)\\
P \boldsymbol{|} 0 &\equiv P\\
\nu x.0 &\equiv 0\\
\nu x.\texttt{Stop} &\equiv \texttt{Stop}\\
\nu x, y.P &\equiv \nu y, x.P\\
P \boldsymbol{|} Q &\equiv Q \boldsymbol{|} P\\
!P &\equiv P \boldsymbol{|}\, !P
\end{aligned}$$

Fig. 3. Structural congruence

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. We sometimes write $x(y)$ instead of $x(y).0$ as well as $\overline{x}y$ instead of $\overline{x}y.0$.

Definition 2.2. *The only reduction rule of the $\Pi_{\texttt{Stop}}$-calculus is the so-called* interaction $\xrightarrow{ia}$ *which is defined in Fig. 2.*

Definition 2.3. *Let P, Q, R be processes and x, y channel names. Structural congruence $\equiv$ is the least congruence satisfying the laws shown in Fig. 3.*

Remark 2.4. We did not include "new" laws for structural congruences on the constant `Stop`, like $\texttt{Stop} \boldsymbol{|} \texttt{Stop}$ equals `Stop`, or even $\texttt{Stop} \boldsymbol{|} P$ equals `Stop`. We did not want to include them, since this would require to develop a lot of theory known from the π-calculus without `Stop` again for the calculus with `Stop`. Our view is that `Stop` is a mechanism to have an easy notion for successful termination, that can be easily replaced by other similar notions (e.g. observing an open input or output as in barbed testing). However, it is easy to prove that those mentioned equations hold *on the semantic level*, e.g. the contextual equivalence $\texttt{Stop} \boldsymbol{|} P \sim_c \texttt{Stop}$ holds (where $\sim_c$ is defined below in Definition 2.10).

As example, consider the communication that sends name y over channel x and then sends u over channel y:

$$(x(z).\overline{z}u.0 \mid \overline{x}y.y(x).0) \xrightarrow{ia} (\overline{z}u.0[y/z] \mid y(x).0) \equiv (\overline{y}u.0 \mid y(x).0) \xrightarrow{ia} (0 \mid 0) \equiv 0$$

Definition 2.5. *A* process context $C \in \Pi_{\mathtt{Stop},C}$ *is a process that has a hole* $[\cdot]$ *at one process position. They are defined by the grammar shown in Fig. 1. With* $C[P]$ *we denote the substitution of the hole in* C *by* P*. The special class of* reduction contexts *is also defined in Fig. 1.*

Definition 2.6. *A* standard reduction $\xrightarrow{sr}$ *is the application of* $\xrightarrow{ia}$ *within a reduction context modulo structural congruence (see Fig. 2). Let* $\xrightarrow{sr,n}$ *denote* n *standard reductions and* $\xrightarrow{sr,*}$ *denotes the reflexive-transitive closure of* $\xrightarrow{sr}$*.*

Definition 2.7. *A process* $P \in \Pi_{\mathtt{Stop}}$ *is* successful*, if* $P \equiv \mathbb{D}[\mathtt{Stop}]$ *for some* $\mathbb{D} \in PCtxt_{\pi}$*.*

A property of interest is whether standard reductions successfully terminate or not. 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 2.8. *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 2.9. The process $P := \nu x, y.(x(z).0 \mid \overline{x}y.\mathtt{Stop}))$ 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 $P' := \nu x, y.(x(z).0 \mid \overline{x}y.0)$ 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 $P'' := \nu x, y.(\overline{x}y.0 \mid x(z).\mathtt{Stop} \mid x(z).0)$ shows that may-convergence and should-convergence are different. There are two standard-reduction possibilities for P'': 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.

Definition 2.10. *For* $P, Q \in \Pi_{\mathtt{Stop}}$ *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*

$$P \leq_{c,\xi} Q \textit{ iff } \forall C \in \mathcal{C} : C[P] \leq_{\xi} C[Q] \quad \textit{and} \quad P \sim_{c,\xi} Q \textit{ iff } P \leq_{c,\xi} Q \wedge Q \leq_{c,\xi} P$$

Contextual equivalence *of* $\Pi_{\mathtt{Stop}}$*-processes is defined as*

$$P \sim_{c} Q \textit{ iff } P \sim_{c,\downarrow} Q \wedge P \sim_{c,\Downarrow} Q.$$

Inspection of the reduction contexts and the $\xrightarrow{ia}$-reduction shows:

Lemma 2.11. *Let P be a $\Pi_{\mathtt{Stop}}$-process s.t. $FN(P) \subseteq \{x_1,\ldots,x_n\}$.*

1. *If $P \xrightarrow{sr,*} P'$ then $\nu x_1,\ldots,\nu x_n.P \xrightarrow{sr,*} \nu x_1,\ldots,x_n.P'$.*
2. *If $\nu x_1,\ldots,\nu x_n.P \xrightarrow{sr,*} P'$ then $P' \equiv \nu x_1,\ldots,x_n.P''$ and $P \xrightarrow{sr,*} P''$*

Lemma 2.12. *Let $\xi \in \{\downarrow,\uparrow,\Downarrow,\Uparrow\}$, P,Q be $\Pi_{\mathtt{Stop}}$-processes. Then $P \leq_{c,\xi} Q$ if, and only if $\forall C \in \mathcal{C}$ such that $C[P]$ and $C[Q]$ are closed: $C[P] \leq_{\xi} C[Q]$.*

Proof. One direction is trivial. For the other direction, we first consider $\xi = \downarrow$. Assume that $\forall C \in \mathcal{C}$ such that $C[P]$ and $C[Q]$ are closed: $C[P] \leq_{\downarrow} C[Q]$. Let C be an arbitrary context such that $C[P]{\downarrow}$ and let $FN(C[P] \mid C[Q]) = \{x_1,\ldots,x_n\}$. Lemma 2.11 shows $\nu x_1,\ldots,x_n.C[P]{\downarrow}$. The precondition shows $\nu x_1,\ldots,\nu x_n.C[Q]{\downarrow}$. Lemma 2.11 shows $C[Q]{\downarrow}$.

The case $\xi = \Uparrow$ holds, since $\Uparrow = \neg\downarrow$: If $\forall C \in \mathcal{C}$ such that $C[P]$ and $C[Q]$ are closed: $C[P] \leq_{\Uparrow} C[Q]$. then this shows $C[Q] \leq_{c,\downarrow} C[P]$ which is equivalent to $C[P] \leq_{c,\Uparrow} C[Q]$.

For $\xi = \uparrow$, assume that $\forall C \in \mathcal{C}$ such that $C[P]$ and $C[Q]$ are closed: $C[P] \leq_{\uparrow} C[Q]$. Let C be an arbitrary context such that $C[P]\uparrow$ and let $FN(C[P]|C[Q]) = \{x_1,\ldots,x_n\}$. Lemma 2.11 and since adding or removing ν-binders does not change success nor must-divergence we have $\nu x_1,\ldots,x_n.C[P]\uparrow$. Now the precondition shows $\nu x_1,\ldots,\nu x_n.C[Q]\uparrow$, and Lemma 2.11 and since adding or removing ν-binders does not change success nor must-divergence shows $C[Q]\uparrow$. Finally, the case $\xi = \Downarrow$ follows by symmetry and since $P\uparrow \iff \neg(P\Downarrow)$.

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

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

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 analogous to the proof given in [23]. for barbed may- and should-testing. The theorem allows us to restrict standard reduction to $\xrightarrow{dsr}$-reduction when reasoning on reduction sequences that witness may-convergence or may-divergence, resp.

Theorem 2.14. *For all processes $P \in \Pi_{\mathtt{Stop}}$ the following holds:*
1. $P{\downarrow}$ iff $P \xrightarrow{dsr,*} \mathbb{D}[\mathtt{Stop}]$. 2. $P{\uparrow}$ iff $\exists P'$ such that $P \xrightarrow{dsr,*} P'$ and $P'{\Uparrow}$.

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

Fig. 4. Syntax of expressions, processes, and types of *CH*

$$
\begin{aligned}
P_1 \mathbin{\boldsymbol{|}} P_2 &\equiv P_2 \mathbin{\boldsymbol{|}} P_1\\
(P_1 \mathbin{\boldsymbol{|}} P_2) \mathbin{\boldsymbol{|}} P_3 &\equiv P_1 \mathbin{\boldsymbol{|}} (P_2 \mathbin{\boldsymbol{|}} P_3)\\
(\nu x.P_1) \mathbin{\boldsymbol{|}} P_2 &\equiv \nu x.(P_1 \mathbin{\boldsymbol{|}} P_2), x {\not\in} FV(P_2)\\
\nu x_1.\nu x_2.P &\equiv \nu x_2.\nu x_1.P\\
P_1 &\equiv P_2 \text{ if } P_1 =_\alpha P_2
\end{aligned}
$$

Fig. 5. Structural congruence of *CH*

$$
\begin{aligned}
\mathbb{D} \in \mathit{PCtxt}_{\mathit{CH}} &::= [\cdot] \mid \mathbb{D} \mathbin{\boldsymbol{|}} P \mid P \mathbin{\boldsymbol{|}} \mathbb{D} \mid \nu x.\mathbb{D}\\
\mathbb{M} \in \mathit{MCtxt}_{\mathit{CH}} &::= [\cdot] \mid \mathbb{M} \mathbin{\mathtt{>\!\!>=}} e\\
\mathbb{F} \in \mathit{FCtxt}_{\mathit{CH}} &::= \mathbb{E} \mid (\mathtt{takeMVar}\, \mathbb{E}) \mid (\mathtt{putMVar}\, \mathbb{E}\, e)\\
\mathbb{E} \in \mathit{ECtxt}_{\mathit{CH}} &::= [\cdot] \mid (\mathbb{E}\, e) \mid (\mathtt{seq}\, \mathbb{E}\, e)\\
&\quad\ \mid (\mathtt{case}\, \mathbb{E}\, \mathtt{of}\, \mathit{alts})
\end{aligned}
$$

Fig. 6. Several context classes

Monadic Computations:

(lunit)	$\Leftarrow \mathbb{M}[\mathtt{return}\ e_1 \mathbin{\mathtt{>\!\!>=}}\ e_2] \xrightarrow{sr} \Leftarrow \mathbb{M}[e_2\ e_1]$
(tmvar)	$\Leftarrow \mathbb{M}[\mathtt{takeMVar}\ x] \mathbin{\boldsymbol{|}} x \,\mathsf{\mathbf{m}}\, e \xrightarrow{sr} \Leftarrow \mathbb{M}[\mathtt{return}\ e] \mathbin{\boldsymbol{|}} x \,\mathsf{\mathbf{m}}\, -$
(pmvar)	$\Leftarrow \mathbb{M}[\mathtt{putMVar}\ x\ e] \mathbin{\boldsymbol{|}} x \,\mathsf{\mathbf{m}}\, - \xrightarrow{sr} \Leftarrow \mathbb{M}[\mathtt{return}\ ()] \mathbin{\boldsymbol{|}} x \,\mathsf{\mathbf{m}}\, e$
(nmvar)	$\Leftarrow \mathbb{M}[\mathtt{newMVar}\ e] \xrightarrow{sr} \nu x.(\Leftarrow \mathbb{M}[\mathtt{return}\ x] \mathbin{\boldsymbol{|}} x \,\mathsf{\mathbf{m}}\, e)$
(fork)	$\Leftarrow \mathbb{M}[\mathtt{forkIO}\ e] \xrightarrow{sr} \Leftarrow \mathbb{M}[\mathtt{return}\ ()] \mathbin{\boldsymbol{|}} \Leftarrow e$

Functional Evaluation:

(cpce)	$\Leftarrow \mathbb{M}[\mathbb{F}[x]] \mathbin{\boldsymbol{|}} x = e \xrightarrow{sr} \Leftarrow \mathbb{M}[\mathbb{F}[e]] \mathbin{\boldsymbol{|}} x = e$
(mkbinds)	$\Leftarrow \mathbb{M}[\mathbb{F}[\mathtt{letrec}\, x_1{=}e_1, \ldots, x_n{=}e_n\, \mathtt{in}\, e]]$
	$\xrightarrow{sr} \nu x_1 \ldots x_n.(\Leftarrow \mathbb{M}[\mathbb{F}[e]] \mathbin{\boldsymbol{|}} x_1{=}e_1 \mathbin{\boldsymbol{|}} \ldots \mathbin{\boldsymbol{|}} x_n{=}e_n)$
(beta)	$\Leftarrow \mathbb{M}[\mathbb{F}[((\lambda x.e_1)\ e_2)]] \xrightarrow{sr} \Leftarrow \mathbb{M}[\mathbb{F}[e_1[e_2/x]]]$
(case)	$\Leftarrow \mathbb{M}[\mathbb{F}[\mathtt{case}_T\ (c\ e_1 \ldots e_n)\ \mathtt{of}\ \ldots (c\ y_1 \ldots y_n\ \mathtt{->}\ e) \ldots]]$
	$\xrightarrow{sr} \Leftarrow \mathbb{M}[\mathbb{F}[e[e_1/y_1, \ldots, e_n/y_n]]]$
(seq)	$\Leftarrow \mathbb{M}[\mathbb{F}[(\mathtt{seq}\ v\ e)]] \xrightarrow{sr} \Leftarrow \mathbb{M}[\mathbb{F}[e]]$ 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. 7. Standard reduction rules of *CH* (call-by-name-version)

3 The Process Calculus *CH*

We introduce the calculus *CH* (which is a variant of the calculus *CHF* in [25,26]) which models a core language of Concurrent Haskell [18]. We assume a partitioned set of *data constructors* c where each family represents a type T. The data constructors of type T are $c_{T,1}, \dots, c_{T,|T|}$ where each $c_{T,i}$ has an arity $\mathrm{ar}(c_{T,i}) \geq 0$. We assume that there is a type `()` with data constructor `()`, a type `Bool` with constructors `True`, `False`, a type `List` with constructors `Nil` and `:` (written infix as in Haskell), and a type `Pair` with a constructor $(,)$ written as (a, b). The syntax of the calculus *CH* has processes $P \in \mathit{Proc}_{CH}$ on the toplayer which may have expressions $e \in \mathit{Expr}_{CH}$ as subterms. The syntax of both is shown in Fig. 4 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 "$\boldsymbol{|}$". The ν-binder restricts the scope of a variable. A concurrent thread $\Leftarrow e$ evaluates the expression e. In a process there is (at most one) unique distinguished thread, called the *main thread* written as $\overset{\mathsf{main}}{\Longleftarrow} e$. 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*. If variable x is a name of an MVar or a binding variable, then x is called an *introduced variable*. An introduced variable is visible to the whole process unless its scope is restricted by a ν-binder, i.e. in $Q \,\boldsymbol{|}\, \nu x.P$ the scope of introduced variable x is process P. A process is *well-formed*, if all introduced variables are pairwise distinct and there exists at most one main thread $\overset{\mathsf{main}}{\Longleftarrow} e$.

Expressions Expr_{CH} consist of functional expressions and monadic expressions MExpr_{CH} that model IO-operations. Functional expressions are variables, *abstractions* $\lambda x.e$, *applications* $(e_1\ e_2)$, `seq`-*expressions* $(\mathtt{seq}\ e_1\ e_2)$, *constructor applications* $(c\ e_1\ \dots\ e_{\mathrm{ar}(c)})$, `letrec`-*expressions* $(\mathtt{letrec}\ x_1 = e_1, \dots, x_n = e_n\ \mathtt{in}\ e)$, and $\mathtt{case}_T$-*expressions* for every type T. We abbreviate `case`-expressions as $\mathtt{case}_T\ e\ \mathtt{of}\ \mathit{alts}$ where *alts* are the `case`-*alternatives*. The `case`-alternatives have exactly one alternative $(c_{T,i}\ x_1 \dots x_{\mathrm{ar}(c_{T,i})}\ \mathtt{->}\ e_i)$ for every constructor $c_{T,i}$ of type T, where $x_1, \dots, x_{\mathrm{ar}(c_{T,i})}$ (occurring in the *pattern* $c_{T,i}\ x_1 \dots x_{\mathrm{ar}(c_{T,i})}$) are pairwise distinct variables that become bound with scope e_i. We omit the type index T in $\mathtt{case}_T$ if it is clear from the context. In $(\mathtt{letrec}\ x_1 = e_1, \dots, x_n = e_n\ \mathtt{in}\ e)$ the variables $x_1, \dots, x_n$ are pairwise distinct and the bindings $x_i = e_i$ are recursive, i.e. the scope of x_i is $e_1, \dots, e_n$ and e. *Monadic operators* `newMVar`, `takeMVar`, and `putMVar` are used to create, to empty and to fill MVars, the "bind"-operator `>>=` implements the sequential composition of IO-operations, the `forkIO`-operator performs thread creation, and `return` lifts expressions to monadic expressions.

Functional values are abstractions and constructor applications. If a monadic expression is of the form $(\mathtt{newMVar}\,e)$, $(\mathtt{takeMVar}\,e)$, $(\mathtt{return}\,e)$, $(e_1 \mathbin{\mathtt{>>=}} e_2)$, $(\mathtt{forkIO}\,e)$, or $(\mathtt{putMVar}\,e_1\,e_2)$, then it is a *monadic value*. A *value* is either a functional or a monadic value.

Abstractions, `letrec`-expressions, `case`-alternatives, and $\nu x.P$ introduce variable binders and thus 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. Structural congruence $\equiv$ of *CH*-processes is the least congruence satisfying the laws in Fig. 5. It allows to treat parallel composition as an associative-commutative operator, to shift ν-binders to the top-level of processes and to α-rename processes.

We assume that expressions and processes are well-typed according to a standard monomorphic type system: Since the typing rules are standard, we omit them, but explain the syntax of types (see Fig. 4): (`IO t`) stands for a monadic action with return type `t`, (`MVar t`) stands for an `MVar` with content type `t`, and $\mathtt{t}_1 \to \mathtt{t}_2$ is a function type. We treat constructors like overloaded constants to use them in a polymorphic way.

We introduce a call-by-name small-step reduction for *CH*. This operational semantics can be shown to be equivalent to (a more realistic) call-by-need semantics. The proof is a copy of the proof given in [25] for the calculus *CHF*. However, the equivalence of the reduction strategies is not important or needed for the current paper. That is why we do not include it.

A context is a process or an expression with a (typed) hole $[\cdot]$. We introduce some classes of contexts in Fig. 6. On the process level there are *process contexts* $PCtxt_{CH}$, on expressions first *monadic contexts* $MCtxt_{CH}$ 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* $\mathbb{E} \in ECtxt_{CH}$ are used, and to enforce the evaluation of the (first) argument of the monadic operators `takeMVar` and `putMVar` the class of *forcing contexts* $\mathbb{F} \in FCtxt_{CH}$ is used.

Definition 3.1. *The standard reduction* $\xrightarrow{sr}$ *is defined by the rules and the closure in Fig. 7. We permit standard reduction only for well-formed processes which are not successful.*

Functional evaluation includes β-reduction (beta), rule (cpce) for copying shared bindings into needed positions, 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 monadic evaluation by applying the first monad law. 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. Rule (fork) spawns a new concurrent thread. A concurrent thread finishes its computation if it is of the form $\Leftarrow$ `return` e (where e is of type `()`).

A *CH*-process P is *successful* if $P \equiv \nu x_1 \ldots \nu x_n.(\overset{\mathsf{main}}{\Longleftarrow} \mathtt{return}\ e \mid P')$ and P is well-formed. 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.

Definition 3.2. *Let P be a CH-process. Process P* may-converges *(written as $P\downarrow$), 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 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 iff 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 CH-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* $\forall \mathbb{D} \in PCtxt_{CH} : \mathbb{D}[P_1]\downarrow \implies \mathbb{D}[P_2]\downarrow$
- $P_1 \leq_{c,\Downarrow} P_2$ *iff* $\forall \mathbb{D} \in PCtxt_{CH} : \mathbb{D}[P_1]\Downarrow \implies \mathbb{D}[P_2]\Downarrow$

*For CH-*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 $e_1 \leq_c e_2 \wedge e_2 \leq_c e_1$.*

The following equivalence will help to prove properties of our translation.

Lemma 3.4. *The relations in Definition 3.3 are unchanged, if we add a closedness restriction as follows. Let $\xi \in \{\downarrow, \Downarrow\}$, then $P_1 \leq_{c,\xi} P_2$ iff $\forall \mathbb{D} \in PCtxt_{CH}$ such that $\mathbb{D}[P_1], \mathbb{D}[P_2]$ are closed: $\mathbb{D}[P_1]\xi \implies \mathbb{D}[P_2]\xi$.*

Proof. One direction is obvious. For the other direction, let $\xi \in \{\downarrow, \Downarrow\}$ and assume that for all $\mathbb{D}$ such that $\mathbb{D}[P_1], \mathbb{D}[P_2]$ are closed: $\mathbb{D}[P_1]\xi \implies \mathbb{D}[P_2]\xi$. Assume that $\mathbb{D}[P_1]\xi$ and $FV(\mathbb{D}[P_1]) \cup FV(\mathbb{D}[P_2]) = \{x_1, \ldots, x_n\}$. Since reductions are applicable with or without ν-binders on the top, we have $\nu x_1, \ldots, x_n.\mathbb{D}[P_1]\xi$ and by the precondition $\nu x_1, \ldots, x_n.\mathbb{D}[P_2]\xi$, since $\nu x_1, \ldots, x_n.\mathbb{D}[\cdot]$ is a *PCtxt*-context. From $\nu x_1, \ldots, x_n.\mathbb{D}[P_2]\xi$ also $\mathbb{D}[P_2]\xi$ follows, since reductions are applicable with or without ν-binders on the top. This shows $P_1 \leq_{c,\xi} P_2$. □

Proposition 3.5. *Let P_1, P_2 be well-formed and $P_1 \equiv P_2$. Then $P_1 \sim_c P_2$.*

A program transformation $\eta \subseteq (Proc_{CH} \times Proc_{CH})$ is *correct* iff $P_1 \ \eta \ P_2 \implies P_1 \sim_c P_2$. In Fig. 8 we define garbage collection (gc) and deterministic variants of (tmvar) and (pmvar).

The following proposition holds, since correctness of transformations can be transferred from the calculus *CHF* [25] by an embedding $\iota : CH \to CHF$, such that $\iota(P) \sim_{c,CHF} \iota(P')$ implies $P \sim_{c,CH} P'$. See Appendix C.3 for the full proof.

Proposition 3.6. *The rules (lunit), (nmvar), (fork), (cpce), (mkbinds), (beta), (case), (seq), (gc), (dtmvar), and (dpmvar) are correct program transformations.*

(gc) $\nu x_1, \ldots, x_n.(P \mid \texttt{Comp}(x_1) \mid \ldots \mid \texttt{Comp}(x_n)) \to P$,
if $\forall 1 \leq i \leq n : x_i \notin FV(P)$ and $\texttt{Comp}(x_i)$ is $x_i = e_i, x_i \, \mathbf{m} \, e_i,$ or $x_i \, \mathbf{m} \, -$

(dtmvar) $\nu x.\mathbb{D}[\Leftarrow \mathbb{M}[\texttt{takeMVar x}] \mid x \, \mathbf{m} \, e] \to \nu x.\mathbb{D}[\Leftarrow \mathbb{M}[\texttt{return } e] \mid x \, \mathbf{m} \, -]$,
if $\forall \mathbb{D}' \in PCtxt_{CH}$ and sr-reductions starting with $\mathbb{D}'[\nu x.(\mathbb{D}[\Leftarrow \mathbb{M}[\texttt{takeMVar } x] \mid x \, \mathbf{m} \, e])]$ the first execution of ($\texttt{takeMVar } x$) is in the shown thread.

(dpmvar) $\nu x.\mathbb{D}[\Leftarrow \mathbb{M}[\texttt{putMVar } x \ e] \mid x \, \mathbf{m} \, -] \to \nu x.\mathbb{D}[\Leftarrow \mathbb{M}[\texttt{return ()}] \mid x \, \mathbf{m} \, e]$,
if $\forall \mathbb{D}' \in PCtxt_{CH}$ and sr-reductions starting with $\mathbb{D}'[\nu x.(\mathbb{D}[\Leftarrow \mathbb{M}[\texttt{putMVar } x \ e] \mid x \, \mathbf{m} \, -])]$ the first execution of ($\texttt{putMVar } x \ e'$) for any e' is in the shown thread.

Fig. 8. Garbage Collection and Deterministic Take und Put

4 The Translation τ_0 with Private Names

We present a translation that uses private names (i.e. new MVars) which uses similar ideas as the translation of the pi-calculus into an asynchronous pi-calculus in [3], which was an early example of using private names within a translation. Later in Sect. 5, we discuss translations and investigations on those translations which do not use private names.

We define a translation τ_0 mapping $\Pi_{\texttt{Stop}}$-processes to CH-processes. It uses the data type `Channel` (with data contructor `Channel`), defined in Haskell-syntax as

$$\texttt{data Channel} = \texttt{Channel (MVar (Channel}, (\texttt{MVar ()}))).$$

The type `Channel` is a recursive data type, which can be initialized with a $\perp$-expression. In the following, we abbreviate the expression ($\texttt{case } e \texttt{ of } (\texttt{Channel } m \texttt{ -> } m)$) as ($unchan\ e$) and we use $a \gg b$ as abbreviation for $a \texttt{>>=} (\lambda\_ .\, b)$ and also use Haskell's do-notation as abbreviation, where

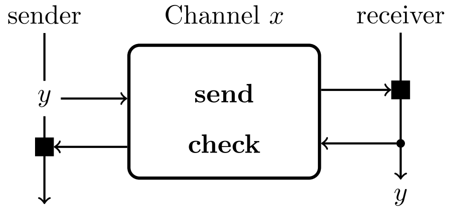
$$\begin{aligned}
&\textbf{do } \{x \leftarrow e_1; e_2\} &&= e_1 \texttt{>>=} \ \lambda x.(\textbf{do } \{e_2\}) \\
&\textbf{do } \{(x, y) \leftarrow e_1; e_2\} &&= e_1 \texttt{>>=} \ \lambda z.\texttt{case}_{\texttt{Pair}} \ z \ \texttt{of} \ (x, y) \texttt{->} (\textbf{do } \{e_2\}) \\
&\textbf{do } \{e_1; e_2\} &&= e_1 \gg (\textbf{do } \{e_2\}), \\
&\textbf{do } \{e\} &&= e
\end{aligned}$$

As a further abbreviation, we write $y \leftarrow \texttt{newEmptyMVar}$ inside a **do**-block to abbreviate the sequence $y \leftarrow \texttt{newMVar } \perp; \texttt{takeMVar } y$, where $\perp$ is a must-divergent expression, for instance $\texttt{letrec } x = x \texttt{ in } x$.

The translation from the $\Pi_{\texttt{Stop}}$-calculus is done by using one MVar per channel which contains a pair consisting of the (translated) name of the channel and a further MVar used for the synchronization. This MVar is private, such that only the sender and the receiver know it. Privacy is established by the sender: it creates a new MVar for every send operation.

$$\begin{array}{ll}
\tau_0(P) & = \overset{\mathsf{main}}{\Longleftarrow} \textbf{do } \{\mathit{stop} \leftarrow \texttt{newMVar}\ ();\texttt{forkIO}\ \tau(P);\texttt{putMVar}\ \mathit{stop}\ ()\} \\
\tau(\overline{x}y.P) & = \textbf{do } \{\mathit{checkx} \leftarrow \texttt{newMVar}\ ();\texttt{putMVar}\ (\mathit{unchan}\ x)\ (y, \mathit{checkx}); \\
 & \qquad \texttt{putMVar}\ \mathit{checkx}\ ();\tau(P)\} \\
\tau(x(y).P) & = \textbf{do } \{(y, \mathit{checkx}) \leftarrow \texttt{takeMVar}\ (\mathit{unchan}\ x);\texttt{takeMVar}\ \mathit{checkx};\tau(P)\}\} \\
\tau(P \mid Q) & = \textbf{do } \{\texttt{forkIO}\ \tau(Q);\tau(P)\} \\
\tau(\nu x.P) & = \textbf{do } \{\mathit{chanx} \leftarrow \texttt{newEmptyMVar};\texttt{letrec}\ x = \texttt{Channel}\ \mathit{chanx}\ \texttt{in}\ \tau(P)\} \\
\tau(0) & = \texttt{return}\ () \\
\tau(\texttt{Stop}) & = \texttt{takeMVar}\ \mathit{stop} \\
\tau(!P) & = \texttt{letrec}\ f = \textbf{do } \{\texttt{forkIO}\ \tau(P); f\}\ \texttt{in}\ f
\end{array}$$

Fig. 9. Translations τ_0 and τ



Message y is sent over channel x by sending a pair (y,check) where *check* is a MVar containing (). The receiver waits (black square) for a message (y, check) by the sender. After sending the message, the sender waits until *check* is emptied, and the receiver acknowledges by emptying the MVar *check*.

Definition 4.1. *We define the translation τ_0 and its inner translation τ from the $\Pi_{\texttt{Stop}}$-calculus into the CH-calculus in Fig. 9 as follows. For contexts, the translations are the same where the context hole is treated like a constant and translated as $\tau([\cdot]) = [\cdot]$.*

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 *CH*-expressions.

Remark 4.2. Except for the main-thread, the translation τ_0 generates a valid Concurrent Haskell-program, i.e. if we write $\tau_0(P) = \overset{\mathsf{main}}{\Longleftarrow} e$ as $\texttt{main} = e$, we can execute the translation in the Haskell-interpreter.

4.1 Properties of the Translation τ_0

In this section we define properties of translations and motivate and present the plan for proving the results for the translations τ_0 and τ, which will be in Sect. 4.2 (and details also in Appendix A).

For the following definition of translation τ being compositional, adequate, or fully abstract, we adopt the view of the translation τ to be a translation from $\Pi_{\texttt{Stop}}$ into the *CH*-language with a special initial evaluation context C_{out}^{τ}.

Definition 4.3. *The context C^{τ}_{out} for translation τ_0 is defined as*

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

We define variants of may- and should-convergence of expressions e within C^{τ}_{out} in CH:

$$e{\downarrow_0} \textit{ iff } C^{\tau}_{out}[e]{\downarrow} \quad \textit{and} \quad e{\Downarrow_0} \textit{ iff } C^{\tau}_{out}[e]{\Downarrow}.$$

We define the relations $\leq_{c,\tau_0}$ and $\sim_{c,\tau_0}$ on CH-expressions:

$$e_1 \leq_{c,\tau_0} e_2 \textit{ iff } \forall C : \textit{if } FV(C[e_1], C[e_2]) \subseteq \{stop\}, \textit{then}$$
$$C[e_1]{\downarrow_0} \implies C[e_2]{\downarrow_0} \textit{ and } C[e_1]{\Downarrow_0} \implies C[e_2]{\Downarrow_0}$$
$$e_1 \sim_{c,\tau_0} e_2 \textit{ iff } e_1 \leq_{c,\tau_0} e_2 \textit{ and } e_1 \leq_{c,\tau_0} e_2.$$

Note that $\leq_{c,CH}$ is a subset of $\leq_{c,\tau_0}$, and hence also $\sim_{c,CH}$ is a subset of $\sim_{c,\tau_0}$ and thus we often can use the more general relations for reasoning and in particular can reuse results from *CH*.

Definition 4.4. *Let $\Pi_{\mathtt{Stop},C}$ be the contexts of $\Pi_{\mathtt{Stop}}$. We define the following properties for τ_0 and τ (see [30,32]). For open processes P, P', we say that translation τ is*

compositional upto $\{\downarrow_0, \Downarrow_0\}$ *iff for all $P \in \Pi_{\mathtt{Stop}}$, all $C \in \Pi_{\mathtt{Stop},C}$, and all $\xi \in \{\downarrow_0, \Downarrow_0\}$: if $FV(C[P]) \subseteq \{stop\}$, then $\tau(C[P])\xi \iff \tau(C)[\tau(P)]\xi$,*
adequate *iff for all processes $P, P' \in \Pi_{\mathtt{Stop}}$: $\tau(P) \leq_{c,\tau_0} \tau(P') \implies P \leq_c P'$, and*
fully abstract *iff for all processes $P, P' \in \Pi_{\mathtt{Stop}}$: $P \leq_c P' \iff \tau(P) \leq_{c,\tau_0} \tau(P')$.*

Convergence equivalence of the translation τ_0 for may- and should-convergence holds. For readability and space reasons the proofs are omitted, but given in a technical appendix. In Appendix A we thus show the following two propositions:

Proposition 4.5. *Let $P \in \Pi_{\mathtt{Stop}}$ be closed. Then τ_0 is convergence-equivalent for $\downarrow$, i.e. $P{\downarrow}$ is equivalent to $\tau_0(P){\downarrow}$. This also implies that $P{\Uparrow}$ is equivalent to $\tau(P){\Uparrow}$.*

Proposition 4.6. *Let $P \in \Pi_{\mathtt{Stop}}$ be closed. Then τ_0 is convergence-equivalent for $\Downarrow$, i.e. $P{\Downarrow}$ is equivalent to $\tau_0(P){\Downarrow}$.*

We informally describe the main steps of the proof. Both propositions have an "if" and an "only-if" part, i.e. to prove them, one has to show four claims:

1. τ_0 preserves may-convergence, i.e. for closed $P \in \Pi_{\mathtt{Stop}}$: $P{\downarrow} \implies \tau_0(P){\downarrow}$.
2. τ_0 reflects may-convergence, i.e. for closed $P \in \Pi_{\mathtt{Stop}}$: $\tau_0(P){\downarrow} \implies P{\downarrow}$.
3. τ_0 preserves must-convergence, i.e. for closed $P \in \Pi_{\mathtt{Stop}}$: $P{\Downarrow} \implies \tau_0(P){\Downarrow}$.
4. τ_0 reflects must-convergence, i.e. for closed $P \in \Pi_{\mathtt{Stop}}$: $\tau_0(P){\Downarrow} \implies P{\Downarrow}$.

Actually, for the latter two parts, we show reflection and preservation of may-divergence, i.e.:

3'. τ_0 reflects may-divergence, i.e. for closed $P \in \Pi_{\texttt{Stop}}$: $\tau_0(P)\uparrow \implies P\uparrow$.
4'. τ_0 preserves may-divergence, i.e. for closed $P \in \Pi_{\texttt{Stop}}$: $P\uparrow \implies \tau_0(P)\uparrow$.

Note that claim 3 is equivalent to 3' and claim 4 is equivalent to claim 4'.

The proof technique for showing parts 1,2,3',4' is to investigate properties of reduction sequences (those that end in a successful process and those that end in a must-divergent process) and to (inductively) apply appropriate (and permitted) rearrangements of reduction sequences and correct program transformations, i.e. those listed in Proposition 3.6.

For the given reduction sequences in the $\Pi_{\texttt{Stop}}$-calculus (giving evidence that $P\downarrow$, or $P\uparrow$ resp. holds) for proving part 1 and 4', we assume that they are given as steps of *dsr*-reductions (Definition 2.13). This simplifies the proof and is correct due to Theorem 2.14.

A further technical detail is that we use another translation σ (and σ_0) which translates several π-process components directly into *CH*-process components (for instance, parallel composition is directly translated into parallel composition), instead of translating them into code which generates the components (as τ does). An intermediate step in the proof is to show equivalence of σ and τ w.r.t. contextual equivalence.

For proving the reflection parts, i.e. part 2 and part 3, the following diagrams sketch the overall idea of the induction proof, which mainly is on the number n of given reductions.

For part 2 the diagram is

$$\begin{tikzcd}
P \arrow[rr, "\sigma_0"] \arrow[d, "{sr,*}"'] & & \sigma_0(P) \arrow[d, "{sr,n}"] \arrow[dl, dashed, "{(sr\cup\sim_c),*}"] \\
P_1 \arrow[r, dashed, "\sigma_0"] & \sigma_0(P_1)(succ.) & Q_1(succ.)
\end{tikzcd}$$

For part 3, the diagram is

$$\begin{tikzcd}
P \arrow[rr, "\sigma_0"] \arrow[d, "{sr,*}"'] & & \sigma_0(P) \arrow[d, "{sr,n}"] \arrow[dl, dashed, "{(sr\cup\sim_c),*}"] \\
P_1\Uparrow \arrow[r, dashed, "\sigma_0"] & \sigma_0(P_1)\Uparrow & Q_1\Uparrow
\end{tikzcd}$$

The difference of both parts is the induction base: For part 2, the processes are successful, while for part 3, the processes have to be must-divergent. Both diagrams also illustrate the use of correct program transformations to obtain a "reordered" sequence of reductions and transformation for $\sigma_0(P)$ such that it can be back-translated into a sequence of sr-reductions in the $\Pi_{\texttt{Stop}}$-calculus.

4.2 Main Results for the Translation τ_0

We show the main result that the translation is adequate (Theorem 4.10). The interpretation of this result is that the pi-calculus with the concurrent semantics is semantically represented within *CH*. This result is on a more abstract level, since it is based on the property whether the programs (in all contexts) produce values or may run into failure, or get stuck; or not. Since the pi-calculus does not have a notion of values, like numbers or lists, also the translated processes cannot be compared w.r.t. values other than a single form of value.

The translation τ_0 is not fully abstract (Theorem 4.11), which is rather natural, since it only means that it is mapped into a subset of the expressions and that this is a proper subset w.r.t. the semantics.

First we show a simple form of a context lemma:

Lemma 4.7. *Let e, e' be CH-expressions, where the only free variable in e, e' is stop. Then $C^{\tau}_{out}[e] \leq_c C^{\tau}_{out}[e']$ iff $C^{\tau}_{out}[e]{\downarrow} \implies C^{\tau}_{out}[e']{\downarrow}$ and $C^{\tau}_{out}[e]{\Downarrow} \implies C^{\tau}_{out}[e']{\Downarrow}$.*

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

1. Let $\mathbb{D}$ be a process-context such that $\mathbb{D}[C^{\tau}_{out}[e]]$ and $\mathbb{D}[C^{\tau}_{out}[e']]$ are closed, 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 *CH*-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. Assume that $\mathbb{D}[C^{\tau}_{out}[e']]{\uparrow}$. We have to show that $\mathbb{D}[C^{\tau}_{out}[e]]{\uparrow}$ holds. Due to closedness, we have $\mathbb{D}[C^{\tau}_{out}[e]] \equiv P_{\mathbb{D}} \mid C^{\tau}_{out}[e]$ 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 4.8. *The translation τ is compositional upto $\{\downarrow_0, \Downarrow_0\}$.*

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

Theorem 4.9. *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 $\tau_0(P){\Downarrow}$. This also shows convergence-equivalence of τ w.r.t. $\downarrow_0, \Downarrow_0$, i.e. For closed $P \in \Pi_{\mathtt{Stop}}$: $P{\downarrow} \iff \tau(P){\downarrow_0}$ and $P{\Downarrow} \iff \tau(P){\Downarrow_0}$.*

Proof. This follows from Proposition 4.5 for may-convergence and from Proposition 4.6 for should-convergence.

We show in the following that the translation τ transports $\Pi_{\mathtt{Stop}}$-processes into *CH*, such that adequacy holds. This is a rather strong statement: It shows that the translated processes also mimic the behavior of the original $\Pi_{\mathtt{Stop}}$-processes when plugged into contexts in a correct way. If the translated open processes cannot be distinguished by $\leq_{c,\tau_0}$, i.e. there is no test that detects a difference where may and should-convergence are the observables, then the original processes are equivalent in the pi-calculus.

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

We state and prove the main result of our paper.

Theorem 4.10. *The translation τ is adequate.*

Proof. We prove the adequacy-property for the preorder $\leq_{c,\tau_0}$. The property for the contextual equivalence $\sim_{c,\tau_0}$ and $\sim_c$ then follows by symmetry. Let P, P' be $\Pi_{\mathtt{Stop}}$-processes, such that $\tau(P) \leq_{c,\tau_0} \tau(P')$. We show that $P \leq_c P'$. Now we use the equivalence in Lemma 3.4 and restrict considerations to closed $C[P], C[P']$ below: Let C be a context in $\Pi_{\mathtt{Stop}}$, such that $C[P], C[P']$ are closed and $C[P]{\downarrow}$. 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 4.8. 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 4.8. Again, closed convergence equivalence implies $C[P']{\downarrow}$. The same arguments holds for $\Downarrow$ instead of $\downarrow$. The computation can be done for every context C that satisfies the conditions.
In summary, we obtain $P \leq_c P'$.

Theorem 4.11. *The translation τ is not fully abstract.*

Proof. This holds, since an open translation can be closed in *CH* by a context without initializing the ν-bound MVars. For $P = \bar{x}(y).\mathtt{Stop} \mid x(z).\mathtt{Stop}$, we have $P \sim_c \mathtt{Stop}$ in the π-calculus (see Example D.2), but $\tau(P) \not\sim_{c,0} \tau(\mathtt{Stop})$ in *CH*: let $\mathbb{D}$ be a process context that does not initialize the MVars for x (as the translation does). Then $\mathbb{D}[\tau(P)]{\Uparrow_0}$, but $\mathbb{D}[\tau(\mathtt{Stop})]{\Downarrow_0}$.

However, restricted to closed processes, full abstraction holds:

Theorem 4.12. *For closed $P_1, P_2 \in \Pi_{\mathtt{Stop}}$, the equivalence $P_1 \leq_c P_2 \iff \tau(P_1) \leq_{c,\tau_0} \tau(P_2)$ holds.*

Proof. For closed P, P', the implication $P \leq_c P' \implies \tau(P_1) \leq_{c,\tau_0} \tau(P_2)$ follows from Lemma 4.7, since τ_0 produces closed processes that are in context C^{τ}_{out}. The other direction follows from Theorem 4.10.

5 Translations with Global Names

In this section, we discuss translations that do not use private names, but only global names. We report on an automated tool that we implemented and that searches for those translations trying to refute their correctness. We only consider the aspect of how to encode the synchronous message passing of the π-calculus, the other aspects (encoding parallel composition, replication and the `Stop`-constant) are not discussed, where we assume that they are encoded as before. We also keep the main idea to translate a channel of the π-calculus into *CH*: we represent a π-channel in *CH* as an object of a user-defined data type `Channel` that consists of

1. an MVar for transferring the message (which again is a `Channel`),
2. additional MVars for implementing a correct synchronization mechanism.

In the translation τ in the previous section, we used a private MVar (which was created by the sender for every communication action and transferred together with the message). In this section we investigate translations where this mechanism is replaced by one or several public MVars, which are created once together with the channel object. To restrict the search space for translations, only the synchronization mechanism of MVars (by emptying and filling them) is used, but we forbid to transfer specific data (like numbers etc.). Hence, we restrict these MVars (which are called *check-MVars* from now on) to be of type `MVar ()`. Such MVars are comparable to binary semaphores, where filling and emptying correspond to signal and wait.

In summary, we now analyze translations of π-calculus channels into a *CH*-data type `Channel` where the data type definition in Haskell is of the form

$$\texttt{data Channel = Chan (MVar Channel) }\underbrace{\texttt{(MVar ())}\ldots\texttt{(MVar ())}}_{n\text{ times}}$$

A channel x of the π-calculus is then represented as a *CH*-binding

$$x = \texttt{Chan}\ \mathit{content}\ \mathit{check}_1\ \ldots\ \mathit{check}_n$$

where $\mathit{content}, \mathit{check}_1, \ldots, \mathit{check}_n$ are appropriately initialized (i.e. empty) MVars. The MVars are not private (but public (or global), since all processes which know the name x have access to the $n+1$ components of the channel.

After fixing this representation of a π-channel in *CH*, the task is to translate the input- and output actions $x(y)$, $\overline{x}z$ as *CH*-programs such that the interaction reduction is performed correct and synchronously[5].

Let us call the translation of $x(y)$, the receiver program (the receiver, for short), and the translation of $\overline{x}z$ the sender program (the sender, for short). To simplify the analysis, we restrict the allowed operations of the sender and the receiver, and thus allow only the following operations:

`putS`**:** The sender puts its message into the *contents*-MVar of the channel. This operation occurs exactly once in the sender program. We write it as $\texttt{putS}_x\ z$, or even more abstractly as `putS`, if x and z are clear from the context. It represents the *CH*-expression $\texttt{case}\ x\ \texttt{of}\ (\texttt{Chan}\ c\ a_1\ \ldots\ a_n\ \texttt{->}\ \texttt{putMVar}\ c\ z \gg e)$ where e is the remaining program of the sender.

`takeS` The receiver takes the message from the *contents*-MVar of channel x and replaces the channel name y by the received name in the subsequent program (which does not belong the current input-operation). This operation occurs exactly once in the receive program. We write this as $\texttt{takeS}_x\ y$, or even more abstractly as `takeS`, if x and y are clear from the context.

[5] We omit the translation of the ν-operator at this moment, but clearly, it has to construct and initialize the appropriate MVars. In Definition 5.2 the full translation will be given.

It represents the *CH*-expression

$$\texttt{case}\ x\ \texttt{of}\ (\texttt{Chan}\ c\ a_1\ \dots\ a_n\ \texttt{->}\ \texttt{takeMVar}\ c\ \texttt{>>=}\ \lambda y.e)$$

where e is the remaining program of the receiver.

In **do**-notation, we also write **do** $\{y \leftarrow \texttt{takeS}_x; e\}$ to abbreviate the above *CH*-expression.

putC and takeC: The sender and the receiver may synchronize on a check-MVar $check_1, \dots, check_n$ by either putting () into it or by emptying the MVar. These operations are written as $\texttt{putC}_x^i$ and $\texttt{takeC}_x^i$, or even more abstractly as $\texttt{putC}^i, \texttt{takeC}^i$ if the name x is clear from the context. We write putC and takeC if there is only one check-MVar.
Let e be the remaining program of the sender or receiver. Then $\texttt{putC}_x^i$ represents the *CH*-program

$$\texttt{case}\ x\ \texttt{of}\ (\texttt{Chan}\ c\ a_1\ \dots\ a_n\ \texttt{->}\ \texttt{putMVar}\ a_i\ ()\ \texttt{>>}\ e)$$

and $\texttt{takeC}_x^i$ represents the *CH*-program

$$\texttt{case}\ x\ \texttt{of}\ (\texttt{Chan}\ c\ a_1\ \dots\ a_n\ \texttt{->}\ \texttt{takeMVar}\ a_i\ \texttt{>>}\ e)$$

We restrict our search for translations to the case that the sender and the receiver programs are sequences of the above operations, and that they are independent of the channel name x. With this restriction we can abstractly write the translation of the sender and the receiver program as a pair of sequences, where only $\texttt{putS}, \texttt{takeS}, \texttt{putC}^i$ and $\texttt{takeC}^i$ operations are used. We make some more restrictions in the following definition:

Definition 5.1. *Let $n > 0$ be a number of check-MVars. A* standard global synchronized-to-buffer translation *(or* gstb-translation*) is represented as a pair $(T_{send}, T_{receive})$ of a send-sequence T_{send} and of a receive-sequence $T_{receive}$ which both consist of* putS*,* takeS*,* $\texttt{putC}^i$ *and* $\texttt{takeC}^i$ *operations, where*

- *The send-sequence contains* putS *(exactly once), and the receive-sequence contains* takeS *(exactly once),*
- *For every* $\texttt{putC}^i$*-action in $(T_{send}, T_{receive})$, there is also a* $\texttt{takeC}^i$*-action in $(T_{send}, T_{receive})$.*
- *We can assume that in the send-sequence the indices i are ascending. I.e. if* $\texttt{putC}^i$ *or* $\texttt{takeC}^i$ *is before* $\texttt{putC}^j$ *or* $\texttt{takeC}^j$*, then $i < j$.*

In the following we often say translation instead of gstb-translation, if this is clear from the context.

Definition 5.2. *Let $T = (T_{send}, T_{receive})$ be a gstb-translation. We write $T_{send}^{x,y}$ for the program T_{send} instantiated for the output-prefix $\overline{x}y$, i.e. the* putS*-operation is* $\texttt{putS}_x\ y$*, and all other operations are indexed with name x, i.e* $\texttt{takeC}^i$ *becomes* $\texttt{takeC}_x^i$ *and* $\texttt{putC}^i$ *becomes* $\texttt{putC}_x^i$*. We write $T_{receive}^{x,y}$ for the program $T_{receive}$ instantiated for the input-prefix $\overline{x}y$, i.e. the* takeS*-operation is* $\texttt{takeS}_x\ y$*, and all* *other operations are indexed with name x, i.e* $\mathtt{takeC}^i$ *becomes* $\mathtt{takeC}^i_x$ *and* $\mathtt{putC}^i$ *becomes* $\mathtt{putC}^i_x$*.*

The induced translations $\phi_{0,T}$ *and* ϕ_T *of* $(T_{send}, T_{receive})$ *are defined as follows, where n is the number of check-MVars used by the translation:*

$$
\begin{array}{ll}
\phi_{0,T}(P) & \overset{\mathsf{main}}{\Longleftarrow} \mathbf{do}\ \{\mathit{stop} \leftarrow \mathtt{newMVar}\ (); \mathtt{forkIO}\ \phi_T(P); \mathtt{putMVar}\ \mathit{stop}\ ()\} \\
\phi_T(\overline{x}y.P) & = \mathbf{do}\ \{T^{x,y}_{send}; \phi_T(P)\} \\
\phi_T(x(y).P) & = \mathbf{do}\ \{T^{x,y}_{receive}; \phi_T(P)\} \\
\phi_T(P \mid Q) & = \mathbf{do}\ \{\mathtt{forkIO}\ \phi_T(Q); \phi_T(P)\} \\
\phi_T(\nu x.P) & = \mathbf{do}\,\{\mathit{contx} \leftarrow \mathtt{newEmptyMVar}; \\
 & \qquad \mathit{checkx}_1 \leftarrow \mathtt{newEmptyMVar}; \ldots; \mathit{checkx}_n \leftarrow \mathtt{newEmptyMVar}; \\
 & \qquad \mathtt{letrec}\ x = \mathtt{Channel}\,\mathit{contx}\ \mathit{checkx}_1 \ldots \mathit{checkx}_n\ \mathtt{in}\ \phi_T(P)\} \\
\phi_T(0) & = \mathtt{return}\ () \\
\phi_T(\mathsf{Stop}) & = \mathtt{takeMVar}\ \mathit{stop} \\
\phi_T(!P) & = \mathtt{letrec}\ f = \mathbf{do}\ \{\mathtt{forkIO}\ \phi_T(P); f\}\ \mathtt{in}\ f
\end{array}
$$

Note that the induced translations are defined similar to the translation τ_0 and τ, where the differences are the representations of the channel, and thus the translation of νx, $x(y)$, and $\overline{x}y$ is changed, while the other cases remain the same. Especially, it holds that $\phi_{0,T}(P) = C^{\tau}_{out}[\phi_T]$. Using the same arguments as in the proof of Theorem 4.10 we can show the following proposition.

Proposition 5.3. *If ϕ_T is closed convergence equivalent, then ϕ_T is adequate and on closed processes it is fully-abstract.*

We also speak of an *execution* of a translation $(T_{send}, T_{receive})$ for name x. Here we mean the simulation of the abstract program, i.e. a program that starts with empty MVars $x, x_1, \ldots, x_n$, and is an interleaved sequence of actions from the send and receive-sequence T_{send} and $T_{receive}$, respectively.

5.1 Classifying Translations

To speak about the translations we make some more classifications:

Number n of check-MVars We classify the translations by the number of check-MVars that are present in the `Channel`-type

Multiple-Uses We say that a translation allows *multiple uses*, if the same check-MVar is used more than once, i.e. the sender and/or receiver may contain $\mathtt{takeC}^i$ and $\mathtt{putC}^i$ more than once for the same i.

Interprocess check restriction We say that a translation has the *interprocess check restriction*, if for every i: $\mathtt{takeC}^i$ and $\mathtt{putC}^i$ do not occur both in T_{send}, and also not in $T_{receive}$.

We define some properties of gstb-translations:

Definition 5.4. *A translation $(T_{send}, T_{receive})$ according to Definition 5.1 is*

- executable, *if there is a deadlock free interleaving execution of the parallel combination of* T_{send} *and* $T_{receive}$.
- communicating, *if* T_{send} *contains at least one* $\mathtt{takeC}^i$*-action.*
- overlap-free, *if for a fixed name x, starting with empty MVars, every interleaved (concurrent) execution of T_{send} and $T_{receive}$ cannot be disturbed by starting another T_{send} and $T_{receive}$. More formally, let $((s_1;...;s_i);(r_1;....r_j))$ and $((s'_1;...;s'_i);(r'_1;....r'_j))$ be two copies of $(T_{send}, T_{receive})$ for a fixed name x. We call a command a_k from one of the four sequences, an a-action for $a \in \{s, s', r, r'\}$. The translation T is overlap-free if every execution of the four sequences has the property that it can be split into a prefix and a suffix (called parts in the following) such that one of the following properties holds*
  - *one part contains only s- and r-actions, and the other part contains only s'- and r'-actions, or*
  - *one part contains only s- and r'-actions and the other part contains only s'- and r-actions.*

5.2 Simulating Translations to Refute their Correctness

We implemented a tool to enumerate translations and then to test whether every translation preserves and reflects may- and should-convergence. The above mentioned parallel execution of T_{send} and $T_{receive}$ is not sufficient to refute most of the translations, since it corresponds to the evaluation of the π-program $\nu x.(x(y) \mid \overline{x}z)$ (which is must-divergent, since no $\mathtt{Stop}$ occurs). Thus our approach is to apply the translation to a subset of π-process for which we can decide may- and should-convergence (before and after the translation) in an automated way. Thus we consider only π-programs of the form $(\nu x_1, \ldots, x_n.P)$ where P contains only 0, $\mathtt{Stop}$, parallel composition $\mid$, and input- and output-prefixes. Thus, the programs are replication free and the ν-binders are on the top, and hence terminate. In the following we omit the ν-binders, but always mean them to be present.

Our simulation tool[6] can execute all possible evaluations of such π-processes and – since all evaluation paths are of finite length, the tool can check for may- and should-convergence of the π-program. For the translated program, we do not generate a full *CH*-program: We generate a sequence of sequences of $\mathtt{takeS}_x, \mathtt{putS}_x, \mathtt{takeC}^i_x, \mathtt{putC}^i_x\ z$ and $\mathtt{Stop}$-operations by applying the translation to all action prefixes in π-program and by translating $\mathtt{Stop}$ into $\mathtt{Stop}$, 0 into an empty sequence. We get a sequence of sequences, since we have several threads and each thread is represented by such a sequence. For executing the translated program we simulate the global store (of MVars) and execute all possible interleavings where we check for may- and should-convergence by looking whether the $\mathtt{Stop}$ eventually occurs at the beginning of the sequence. This quite obviously simulates the behavior of the real *CH*-program but in a controllable manner.

Now having an encoding of the sender- and the receiver program, we use a π-calculus process P and

[6] Available via https://gitlab.com/davidsabel/refute-pi.

1. Translate it with the encodings in the sequence of sequences consisting of $\mathtt{takeS}_x, \mathtt{putS}_x, \mathtt{takeC}^i_x, \mathtt{putC}^i_x\ z$ and $\mathtt{Stop}$-operations.
2. Simulate the execution on all interleavings
3. Test may- and should convergence of the original π-program P as well as the encoded program (w.r.t. the simulation)
4. Compare the convergence before and after the translation. If there is a difference in the convergence behavior, then P is a counter-example for the correctness of the encodings.

5.3 Translations Without Check-MVars

The (trivial) case that there is no check-MVar leads to one possible translation $([\mathtt{putS}], [\mathtt{takeS}])$ which means that $\overline{x}z$ is translated into $\mathtt{putS}_x\ y$ and $x(y)$ is translated into $\mathtt{takeS}_x\ y$. This translation is not correct, since for instance the π-process $\overline{x}z.x(y).\mathtt{Stop}$ is neither may- nor should-convergent, but the translation $\mathtt{putS}_x\ z; \mathtt{takeS}_x\ y; \mathtt{Stop}$ is may- and should-convergent, since the put- and the take-operation can be done sequentially.

5.4 Translations With Interprocess Check Restriction

We first consider translations with the interprocess check restriction. In this case the number of different translations is $n! * 2^n * (n+1)^2$ (where n is the number of check-MVars), which for $n = 1$ results in 8, for $n = 2$ results in 72, and for $n = 3$ in 768 translations.

For a single check-MVar all 8 translations are rejected by our simulation, Table 1 shows the translations and the obtained counter examples.

		may-conv	should-conv	may-conv	should-conv
Translation (Sender,Receiver)	Counter-example (π-process)	before		after	
$([\mathtt{putC}, \mathtt{putS}], [\mathtt{takeC}, \mathtt{takeS}])$	$\overline{x}y.x(y).\mathtt{Stop}$	N	N	Y	Y
$([\mathtt{putC}, \mathtt{putS}], [\mathtt{takeS}, \mathtt{takeC}])$	$\overline{x}y.x(y).\mathtt{Stop}$	N	N	Y	Y
$([\mathtt{putS}, \mathtt{putC}], [\mathtt{takeC}, \mathtt{takeS}])$	$\overline{x}y.x(y).\mathtt{Stop}$	N	N	Y	Y
$([\mathtt{putS}, \mathtt{putC}], [\mathtt{takeS}, \mathtt{takeC}])$	$\overline{x}y.x(y).\mathtt{Stop}$	N	N	Y	Y
$([\mathtt{takeC}, \mathtt{putS}], [\mathtt{putC}, \mathtt{takeS}])$	$\overline{x}y.x(z).\mathtt{Stop} \mid x(w)$	N	N	Y	N
$([\mathtt{takeC}, \mathtt{putS}], [\mathtt{takeS}, \mathtt{putC}])$	$\overline{x}y.\mathtt{Stop} \mid x(y)$	Y	Y	N	N
$([\mathtt{putS}, \mathtt{takeC}], [\mathtt{putC}, \mathtt{takeS}])$	$\overline{x}y.x(z).\mathtt{Stop} \mid x(w)$	N	N	Y	N
$([\mathtt{putS}, \mathtt{takeC}], [\mathtt{takeS}, \mathtt{putC}])$	$\overline{x}z.\overline{z}a.\mathtt{Stop} \mid \overline{x}w.\overline{w}a.\mathtt{Stop} \mid x(y).y(u)$	Y	Y	Y	N

Table 1. Translations using one check-MVar and with the interprocess check restriction

Remark 5.5. We exhibit an example, why the translation

$$(T_{send}, T_{receive}) = ([\mathtt{putS}, \mathtt{takeC}], [\mathtt{takeS}, \mathtt{putC}])$$

is not correct. Consider the following $\Pi_{\mathtt{Stop}}$-process

$$P = \nu x, z, w, a.(\overline{x}z.\overline{z}a.\mathtt{Stop} \mid \overline{x}w.\overline{w}a.\mathtt{Stop} \mid x(y).y(w).0)$$

which is should-convergent, since the two possible reduction sequences starting with P end in successful processes.

Consider the translated *CH*-program where we use the commands abbreviating the concrete *CH*-programs, and where we assume that the program code for creating the MVars and the bindings for the `Channel`-objects are already executed:

$$\begin{array}{l}
\nu \ldots x = \mathtt{Chan}\ c_x\ chk_x \mid z = \mathtt{Chan}\ c_z\ chk_z \mid w = \mathtt{Chan}\ c_w\ chk_w \mid \ldots \\
\mid c_x\,\mathbf{m}\,- \mid chk_x\,\mathbf{m}\,- \mid c_z\,\mathbf{m}\,- \mid chk_z\,\mathbf{m}\,- \mid c_w\,\mathbf{m}\,- \mid chk_w\,\mathbf{m}\,- \mid \ldots \\
\mid \ \mathbf{do}\{\mathtt{putS}_x\ z; \mathtt{takeC}_x; \mathtt{putS}_z\ a; \mathtt{takeC}_z; \ldots\} \\
\mid \ \mathbf{do}\{\mathtt{putS}_x\ w; \mathtt{takeC}_x; \mathtt{putS}_w\ a; \mathtt{takeC}_w; \ldots\} \\
\mid \ \mathbf{do}\{y \leftarrow \mathtt{takeS}_x; \mathtt{putC}_x; w \leftarrow \mathtt{takeS}_y; \ldots\}
\end{array}$$

This reduces after executing $\mathtt{putS}_x\ z$ and $y \leftarrow \mathtt{takeS}_x$

$$\begin{array}{l}
\nu \ldots x = \mathtt{Chan}\ c_x\ chk_x \mid z = \mathtt{Chan}\ c_z\ chk_z \mid w = \mathtt{Chan}\ c_w\ chk_w \mid \ldots \\
\mid c_x\,\mathbf{m}\,- \mid chk_x\,\mathbf{m}\,- \mid c_z\,\mathbf{m}\,- \mid chk_z\,\mathbf{m}\,- \mid c_w\,\mathbf{m}\,- \mid chk_w\,\mathbf{m}\,- \mid \ldots \\
\mid \ \mathbf{do}\{\mathtt{takeC}_x; \mathtt{putS}_z\ a; \mathtt{takeC}_z; \ldots\} \\
\mid \ \mathbf{do}\{\mathtt{putS}_x\ w; \mathtt{takeC}_x; \mathtt{putS}_w\ a; \mathtt{takeC}_w; \ldots\} \\
\mid \ \mathbf{do}\{\mathtt{putC}_x; w \leftarrow \mathtt{takeS}_z; \ldots\}
\end{array}$$

The second and third thread make steps:

$$\begin{array}{l}
\nu \ldots x = \mathtt{Chan}\ c_x\ chk_x \mid z = \mathtt{Chan}\ c_z\ chk_z \mid w = \mathtt{Chan}\ c_w\ chk_w \mid \ldots \\
\mid c_x\,\mathbf{m}\,w \mid chk_x\,\mathbf{m}\,() \mid c_z\,\mathbf{m}\,- \mid chk_z\,\mathbf{m}\,- \mid c_w\,\mathbf{m}\,- \mid chk_w\,\mathbf{m}\,- \mid \ldots \\
\mid \ \mathbf{do}\{\mathtt{takeC}_x; \mathtt{putS}_z\ a; \mathtt{takeC}_z; \ldots\} \\
\mid \ \mathbf{do}\{\mathtt{takeC}_x; \mathtt{putS}_w\ a; \mathtt{takeC}_w; \ldots\} \\
\mid \ \mathbf{do}\{w \leftarrow \mathtt{takeS}_z; \ldots\}
\end{array}$$

The second thread can make an unexpected step:

$$\begin{array}{l}
\nu \ldots x = \mathtt{Chan}\ c_x\ chk_x \mid z = \mathtt{Chan}\ c_z\ chk_z \mid w = \mathtt{Chan}\ c_w\ chk_w \mid \ldots \\
\mid c_x\,\mathbf{m}\,w \mid chk_x\,\mathbf{m}\,- \mid c_z\,\mathbf{m}\,- \mid chk_z\,\mathbf{m}\,- \mid c_w\,\mathbf{m}\,- \mid chk_w\,\mathbf{m}\,- \mid \ldots \\
\mid \ \mathbf{do}\{\mathtt{takeC}_x; \mathtt{putS}_z\ a; \mathtt{takeC}_z; \ldots\} \\
\mid \ \mathbf{do}\{\mathtt{putS}_w\ a; \mathtt{takeC}_w; \ldots\} \\
\mid \ \mathbf{do}\{w \leftarrow \mathtt{takeS}_z; \ldots\}
\end{array}$$

Now the only possible step is

$$\begin{array}{l}
\nu \ldots x = \mathtt{Chan}\ c_x\ chk_x \mid z = \mathtt{Chan}\ c_z\ chk_z \mid w = \mathtt{Chan}\ c_w\ chk_w \mid \ldots \\
\mid c_x\,\mathbf{m}\,w \mid chk_x\,\mathbf{m}\,- \mid c_z\,\mathbf{m}\,- \mid chk_z\,\mathbf{m}\,- \mid c_w\,\mathbf{m}\,a \mid chk_w\,\mathbf{m}\,- \mid \ldots \\
\mid \ \mathbf{do}\{\mathtt{takeC}_x; \mathtt{putS}_z\ a; \mathtt{takeC}_z; \ldots\} \\
\mid \ \mathbf{do}\{\mathtt{takeC}_w; \ldots\} \\
\mid \ \mathbf{do}\{w \leftarrow \mathtt{takeS}_z; \ldots\}
\end{array}$$

and now the process is stuck.

If we use 2 check-MVars, then there are 72 translations. Again our tool is able to refute all of them. Compared to Table 1, there are two further π-processes used, which act as counterexample.

However, one can also use the processes $\overline{x}y.\texttt{Stop} \mid x(y)$ and $\overline{x}y.x(z).\overline{z}q \mid x(z) \mid x(z) \mid \overline{x}z \mid y(u).\texttt{Stop}$ to refute all 72 translations.

For 3 MVars, there are 768 translations, where 762 are rejected (where the counter examples are the same as for 2-check-MVars) and 6 potentially correct translations remain. The six translations are shown in Table 2.

Translations	Sender	Receiver
Translations without potential overlaps		
$\mathfrak{T}_1$	$[\texttt{putS}, \texttt{putC}^1, \texttt{takeC}^2, \texttt{putC}^3]$	$[\texttt{takeC}^1, \texttt{putC}^2, \texttt{takeC}^3, \texttt{takeS}]$
$\mathfrak{T}_2$	$[\texttt{takeC}^1, \texttt{putS}, \texttt{takeC}^2, \texttt{takeC}^3]$	$[\texttt{putC}^3, \texttt{putC}^1, \texttt{takeS}, \texttt{putC}^2]$
$\mathfrak{T}_3$	$[\texttt{putC}^1, \texttt{putS}, \texttt{takeC}^2, \texttt{putC}^3]$	$[\texttt{takeS}, \texttt{putC}^2, \texttt{takeC}^3, \texttt{takeC}^1]$
$\mathfrak{T}_4$	$[\texttt{putC}^1, \texttt{putC}^2, \texttt{takeC}^3, \texttt{putS}]$	$[\texttt{takeC}^2, \texttt{putC}^3, \texttt{takeS}, \texttt{takeC}^1]$
Translations with potential overlaps		
$\mathfrak{T}_5$	$[\texttt{takeC}^1, \texttt{putS}, \texttt{takeC}^2, \texttt{takeC}^3]$	$[\texttt{putC}^1, \texttt{putC}^2, \texttt{takeS}, \texttt{putC}^3]$
$\mathfrak{T}_6$	$[\texttt{putC}^1, \texttt{takeC}^2, \texttt{putS}, \texttt{takeC}^3]$	$[\texttt{takeC}^1, \texttt{putC}^2, \texttt{takeS}, \texttt{putC}^3]$

Table 2. Non-rejected translations for 3 check-MVars and interprocess check restriction

Theorem 5.6. *There is no valid gstb-translation with the interprocess check restriction for less than three check-MVars.*

Proof. (Sketch) Our simulator refuted all translations for less than three check-MVars: All 72 possibilities of translations using 2 check-MVars could be refuted, where two processes suffice to refute all translation variants: $\overline{x}y.\texttt{Stop} \mid x(y)$ and $\overline{x}y.x(z).\overline{z}q \mid x(z) \mid x(z) \mid \overline{x}z \mid y(u).\texttt{Stop}$, where the first process is should-convergent and the latter is neither may- nor should-convergent.

A reason for the failure of translations with less then three check-MVars may be the following result:

Theorem 5.7. *There is no executable, communicating, and overlap-free gstb-translation with the interprocess check restriction for $n < 3$.*

Proof. For $n = 1$ we check the last four translations in Table 1. The first four are non-communicating. For $([\texttt{putS}, \texttt{takeC}], [\texttt{takeS}, \texttt{putC}])$: after executing $\texttt{putS}$,$\texttt{takeS}$ we can execute $\texttt{putS}$ again For $([\texttt{putS}, \texttt{takeC}], [\texttt{putC}, \texttt{takeS}])$: after $\texttt{putC}, \texttt{putS}, \texttt{takeC}$ we can execute $\texttt{putC}$ again. For $([\texttt{takeC}, \texttt{putS}], [\texttt{takeS}, \texttt{putC}])$: A deadlock occurs immediately. Thus the translation is not executable. For $([\texttt{takeC}, \texttt{putS}], [\texttt{putC}, \texttt{takeS}])$: After executing $\texttt{putC}, \texttt{takeC}$, we can execute $\texttt{putC}$ again. For $n = 2$, we have checked all cases using our simulator, and no executable, communicating, and overlap-free translation for $n = 2$ is found. The numbers are: 18 translations are ruled out, since these are non-communicating, 21 lead to a deadlock, and 33 may lead to an overlap. This check could also be done by hand by scanning all 72 translations.

Proposition 5.8. *The first four translations in Table 2 are executable, communicating, and overlap-free, whereas the next two translations are executable, communicating, but are overlapping.*

Proof. The executability and overlap-freeness can be easily checked by simulating the possible executions. In any case, only if all 8 actions are finished, the next send or receive can start.

For the next two translations the overlap shows up after executing $\texttt{putC}^1$, $\texttt{takeC}^1$, since we can again execute $\texttt{putC}^1$. □

In Appendix B.3 we sketch the arguments that the induced translation from the first gstb-translation from Table 2, i.e. $\phi_{\mathfrak{T}_1} =$

$$(T_{send}, T_{receive}) = ([\texttt{putS}, \texttt{putC}^1, \texttt{takeC}^2, \texttt{putC}^3], [\texttt{takeC}^1, \texttt{putC}^2, \texttt{takeC}^3, \texttt{takeS}]),$$

is indeed correct and leaves may- and should convergence invariant. The main help in reasoning is that there is no unintended interleaving of send and receive sequences according to Proposition 5.8. Application of Proposition 5.3 shows the following theorem.

Theorem 5.9. *Translation $\phi_{\mathfrak{T}_1}$ is adequate and on closed processes it is fully-abstract.*

For 4 MVars there are 9600 translations and our simulation refutes 9266 and thus there remain 334 candidates for correct translations.

5.5 Dropping the Interprocess Check Restriction

If we drop the interprocess check restriction, but only allow one check-MVar without reuse, then we get 20 candidates for translations, which are all refuted by our simulation.

Allowing reuse of the single check-MVar seems not to help to construct a correct translation: We simulated this for up to 6 uses (which leads to 420420 candidates for a correct translation) – our simulation refutes all of them.

We conjecture that there is no correct translation for one check-MVar where re-uses are permitted, and where the interprocess check restriction is dropped.

For two MVars and one use but without the interprocess check-MVar there are 420 translations. Our simulation refutes all but two translations, which are shown in Table 3. In the translation $\mathfrak{T}_7$ the second check-MVar is used as a mutex for the senders, ensuring that only one sender can operate at a time, while the translation $\mathfrak{T}_8$ does the same on the receiver side.

Proposition 5.10. *The translations $\mathfrak{T}_7$, $\mathfrak{T}_8$ in Table 3 are executable, communicating, and overlap-free.*

Translation	Sender	Receiver
$\mathfrak{T}_7$	$[\mathtt{putC}^1, \mathtt{putS}, \mathtt{takeC}^2, \mathtt{takeC}^1]$	$[\mathtt{takeS}, \mathtt{putC}^2]$
$\mathfrak{T}_8$	$[\mathtt{takeC}^1, \mathtt{putS}]$	$[\mathtt{putC}^2, \mathtt{putC}^1, \mathtt{takeS}, \mathtt{takeC}^2]$

Table 3. Non-refuted translations for two check-MVars, no interprocess check restriction

Proof. It is easy to verify that the translations are executable and communicating. For the first translation, assume that $\mathtt{putC}^1, \mathtt{putS}$ and $\mathtt{takeS}$ are performed in this order (no other order is possible). Then an additional sender cannot execute its first command before the original sender performs $\mathtt{takeC}^1$ and this again is only possible after the receiver program is finished. An additional receiver can only be executed after a $\mathtt{putS}$ is performed, which cannot be done by the current sender and receivers.

For the second translation, assume that $\mathtt{putC}^2, \mathtt{putC}^1$ and $\mathtt{takeC}^1$ are performed in this order (no other order is possible). An additional receiver can only start after $\mathtt{takeC}^2$ was executed by the original receiver, which can only occur after the original sender and receiver program are fully evaluated. An additional sender can only start after $\mathtt{putC}^1$ has been executed again, but the current sender and receiver cannot execute this command.

In the appendix we sketch in Theorem B.4 that the induced translation from the gstb-translation $\mathfrak{T}_7$, i.e. is (closed) convergence equivalent.

Applying Proposition 5.3 this shows

Theorem 5.11. *Translation $\phi_{\mathfrak{T}_7}$ is adequate and on closed processes it is fully-abstract.*

We are convinced that the same holds for the second translation. We conclude the statistics of our search for translations without the interprocess restriction:

- For 3 MVars there are 10080 translations and 9992 are refuted, i.e. 98 translations are potentially correct. Among the 98 translations, there is the translation $([\mathtt{putC}^1, \mathtt{putS}, \mathtt{takeC}^2, \mathtt{takeC}^1], [\mathtt{putC}^3, \mathtt{takeS}, \mathtt{putC}^2, \mathtt{takeC}^3])$ which is quite intuitive: check-MVar 1 is used as a mutex for all senders on the same channel, check-MVar 3 is used as a mutex for all receivers, and check-MVar 2 is used to acknowledge that the message was received.
- For 4 MVars there are 277200 translations and 273210 are refuted, i.e. 3990 translations are potentially correct.

6 Discussion and Conclusion

We have investigated the problem of translating the π-calculus into the *CH*-calculus modeling Concurrent Haskell and proved that there is a correct translation τ_0 from the π-calculus into *CH*, using private names for the storage of every translated communication construct. This translation is an embedding for closed processes w.r.t. $\sim_c$, and preserves the may-convergence behavior as well as the should-convergence behavior. The translation τ_0 meets the gold standard, since even for open processes, it is adequate w.r.t. the respective contextual equivalences.

For translations that use global names, we started an investigation on exhibiting (potentially) correct translations. The result is that in the design space of all (global) translations we identified several minimal translations that are potentially correct. We characterized all incorrect "small" translations, where we left the issue open whether there is a correct translation with only one check-MVar with unrestricted usage.

For two particular global translations, we have shown that they are convergence equivalent, i.e. they preserve and reflect the may-convergence behavior and the should-convergence behavior on closed processes. We also proved adequacy of the translations (on open processes) and full-abstraction if closed processes are translated. The exact form of the translations were found by our implemented tool to search for translations and to refute their correctness. The tool also showed that there is no correct gstb-translation with the interprocess check restriction for less than 3 check-MVars.

Further work may be to 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 mixed sums or (name) matching operators. For name matching operators, our current translation would require to test usual bindings in *CH* for equality which is not available in core-Haskell. Solutions may either use an adapted translation or an adapted core language that supports so-called observable sharing [6,8]. The translation of mixed-sums into *CH* appears to require more complex translations, where the send and receive-parts are not linear lists of actions.

Another interesting research question for future work is whether the pi-calculus can be embedded in call-by-need lambda calculi with `amb`. Note that [4] show that the `amb`-operator is encodable in the π-calculus.

Acknowledgments

We thank an anonymous reviewer for advises to improve the translation. In particular for providing the counter-example in the last row of Table 1.

References

1. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The Spi calculus. In *CCS 1997*, pages 36–47. ACM, 1997.
2. R. Banach, J. Balázs, and G. A. Papadopoulos. A translation of the pi-calculus into MONSTR. *J.UCS*, 1(6):339–398, 1995.
3. G. Boudol. Asynchrony and the pi-calculus. Technical Report Research Report RR-1702,inria-00076939, INRIA, France, 1992.
4. 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.
5. A. Chaudhuri. A concurrent ML library in concurrent haskell. In G. Hutton and A. P. Tolmach, editors, *ICFP 2009*, pages 269–280. ACM, 2009.
6. K. Claessen and D. Sands. Observable sharing for functional circuit description. In *ASIAN 1999*, volume 1742 of *Lecture Notes in Comput. Sci.*, pages 62–73. Springer, 1999.
7. C. Fournet and G. Gonthier. The join calculus: A language for distributed mobile programming. In *APPSEM 2000*, volume 2395 of *LNCS*, pages 268–332. Springer, 2002.
8. A. Gill. Type-safe observable sharing in haskell. In *Haskell 2009*, pages 117–128. ACM, 2009.
9. Haskell-Community. Haskell main website, 2019. `www.haskell.org`.
10. K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '91, pages 133–147, London, UK, UK, 1991. Springer-Verlag.
11. C. Laneve. On testing equivalence: May and must testing in the join-calculus. Technical Report UBLCS 96-04, University of Bologna, 1996.
12. J. McCarthy. A Basis for a Mathematical Theory of Computation. In *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.
13. R. Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999.
14. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I & II. *Inform. and Comput.*, 100(1):1–77, 1992.
15. 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.
16. D. A. Orchard and N. Yoshida. Effects as sessions, sessions as effects. In *POPL 2016*, pages 568–581. ACM, 2016.
17. C. Palamidessi. Comparing the expressive power of the synchronous and asynchronous pi-calculi. *Math. Structures Comput. Sci.*, 13(5):685–719, 2003.
18. S. L. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *POPL 1996*, pages 295–308. ACM, 1996.
19. G. D. Plotkin. LCF considered as a programming language. *Theor. Comput. Sci.*, 5(3):223–255, 1977.
20. C. Priami. Stochastic pi-calculus. *Comput. J.*, 38(7):578–589, 1995.
21. R. Pucella and P. Panangaden. On the expressive power of first-order boolean functions in PCF. *Theor. Comput. Sci.*, 266(1-2):543–567, 2001.
22. G. Russell. Events in haskell, and how to implement them. In B. C. Pierce, editor, *ICFP 2001*, pages 157–168. ACM, 2001.
23. D. Sabel. Structural Rewriting in the pi-Calculus. In *WPTE 2014*, volume 40 of *OASIcs*, pages 51–62. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014.

24. 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.
25. D. Sabel and M. Schmidt-Schauß. A contextual semantics for Concurrent Haskell with futures. In *PPDP 2011*, pages 101–112. ACM, 2011.
26. D. Sabel and M. Schmidt-Schauß. Conservative concurrency in Haskell. In *LICS 2012*, pages 561–570. IEEE, 2012.
27. D. Sabel and M. Schmidt-Schauß. Observing success in the pi-calculus. In *WPTE 2015*, volume 46 of *OASICS*, pages 31–46, 2015.
28. D. Sangiorgi and D. Walker. On barbed equivalences in pi-calculus. In *CONCUR 200*, volume 2154 of *LNCS*, pages 292–304. Springer, 2001.
29. D. Sangiorgi and D. Walker. *The π-calculus: a theory of mobile processes.* Cambridge university press, 2001.
30. M. Schmidt-Schauß, J. Niehren, J. Schwinghammer, and D. Sabel. Adequacy of compositional translations for observational semantics. In *IFIP TCS 2008*, volume 273 of *IFIP*, pages 521–535. Springer, 2008.
31. M. Schmidt-Schauß, D. Sabel, and N. Dallmeyer. Sequential and parallel improvements in a concurrent functional programming language. In D. Sabel and P. Thiemann, editors, *Proc. PPDP 2018,*, pages 20:1–20:13. ACM, 2018.
32. M. Schmidt-Schauß, D. Sabel, J. Niehren, and J. Schwinghammer. Observational program calculi and the correctness of translations. *Theor. Comput. Sci.*, 577:98–124, 2015.
33. J. Schwinghammer, D. Sabel, M. Schmidt-Schauß, and J. Niehren. Correctly translating concurrency primitives. In *ML 2009*, pages 27–38. ACM, 2009.
34. P. Yang, C. R. Ramakrishnan, and S. A. Smolka. A logical encoding of the pi-calculus: model checking mobile processes using tabled resolution. *STTT*, 6(1):38–66, 2004.

$$\begin{array}{ll}
\sigma_0(P) & = \nu stop.(\overset{\mathsf{main}}{\Longleftarrow} \texttt{putMVar}\ stop\ ()\ \mid\ stop\,\mathbf{m}\,()\ \mid\ \sigma(P)) \\
\sigma(\nu x.P) & = \nu x, chanx.(chanx\,\mathbf{m} - \mid x = \texttt{Channel}\ chanx\ \mid \sigma(P)) \\
\sigma(\overline{x}y.P) & = \Leftarrow \tau(\overline{x}y.P) \\
\sigma(x(y).P) & = \Leftarrow \tau(x(y).P) \\
\sigma(0) & = \Leftarrow \texttt{return ()} \\
\sigma(\texttt{Stop}) & = \Leftarrow \texttt{takeMVar}\ stop \\
\sigma(P \mid Q) & = \sigma(P) \mid \sigma(Q) \\
\sigma(!P) & = \nu f.(\Leftarrow f \mid f = \mathbf{do}\ \{\texttt{forkIO}\ \tau(P); f\})
\end{array}$$

Fig. 10. Translations σ and σ_0

A Convergence Properties of the Translation τ_0

A.1 A Top-Down Variant σ of τ

We define a variant σ of the translation τ. It is intended to simplify the proofs. Instead of generating *CH*-programs that during their run generate concurrent processes and MVars (as τ does), the translation σ generates *CH*-processes, which is closer to a direct implementation. The translation σ only modifies a top-part, and refers to τ for the deeper translation parts that generate expressions. The main reason for introducing the translation σ is that it simplifies the proofs of correctness (of translations τ and τ_0).

Definition A.1. *The translation σ_0 is similar to τ_0, and the translation σ also generates processes. Both translations are defined in Fig. 10. The translations of contexts are defined analogously where the context hole is translated into the context hole, i.e. $\sigma([\cdot]) = [\cdot]$.*

Definition A.2. *The context C^{σ}_{out} for translation σ_0 is defined as*

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

The following strong relationship between the translations τ_0 and σ_0 (τ and σ, resp.) holds and is helpful in proofs for the translation τ:

Lemma A.3. *For all $P \in \Pi_{\texttt{Stop}}$ i) $\tau_0(P) \xrightarrow{sr,*} \sigma_0(P)$ and ii) $\Leftarrow \tau(P) \xrightarrow{sr,*} \sigma(P)$.*

Proof. For the first part, it suffices to verify that $\tau_0(P)$ is a process that reduces to $\sigma_0(P)$ as follows:

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

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 process 0, then $\Leftarrow\tau(P) = \Leftarrow\mathtt{return}\ () = \sigma(P)$. If P is a parallel composition $P_1 \mid P_2$, then

$$\begin{aligned}\Leftarrow\tau(P_1 \mid P_2) &= \Leftarrow\mathbf{do}\ \{\mathtt{forkIO}\ \tau(P_2); \tau(P_1)\}\\ &\xrightarrow{sr,fork}\xrightarrow{sr,beta} \Leftarrow\tau(P_2) \mid \Leftarrow\tau(P_1) \xrightarrow{sr,*} \sigma(P_2) \mid \sigma(P_1) = \sigma(P_1 \mid P_2)\end{aligned}$$

The final $\xrightarrow{sr,*}$-sequence obtained as follows: by the induction hypothesis we have $\Leftarrow\tau(P_i) \xrightarrow{sr,*} \sigma(P_i)$ for $i = 1, 2$ by combing and processing them sequentially we 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

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

Note that the (tmvar)-reduction is deterministic (i.e. a (dtmvar)-transformation), since there is no alternative, and 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

$$\begin{aligned}\Leftarrow\tau(!P) &= \Leftarrow\mathtt{letrec}\ f = \mathbf{do}\ \{\mathtt{forkIO}\ \tau(P); f\}\ \mathtt{in}\ f\\ &\xrightarrow{sr,mkbinds} \nu f.\Leftarrow f \mid f = \mathbf{do}\ \{\mathtt{forkIO}\ \tau(P); f\} = \sigma(!P)\end{aligned}$$

This finishes the induction proof. □

Lemma A.3 and correctness of (fork), (nmvar), (dtmvar), (beta), (cpce), and (mkbinds) (see Proposition 3.6) imply:

Proposition A.4. *For all $P \in \Pi_{\mathtt{Stop}}$: 1. $\sigma(P) \sim_c \Leftarrow\tau(P)$ and 2. $\sigma_0(P) \sim_c \tau_0(P)$. Hence also $\sigma(P) \sim_{c,\tau_0} \Leftarrow\tau(P)$ and $\sigma_0(P) \sim_{c,\tau_0} \tau_0(P)$.*

A.2 Preservation of May-Convergence

The goal of this section is to show that may-convergence of a closed $\Pi_{\mathtt{Stop}}$-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 $\Pi_{\mathtt{Stop}}$-processes P, P', there are corresponding reduction steps of $\sigma_0(P)$ in the CH-calculus.

Lemma A.5. *Let $P \in \Pi_{\texttt{Stop}}$ be a closed process with $P \xrightarrow{dia} P'$. Then there is a sequence $\sigma_0(P) \xrightarrow{sr,*} \sigma_0(P') \mid G$ in the CH-calculus, where G is a closed process that can be seen as garbage, i.e. $\sigma_0(P') \mid G \xrightarrow{gc} \sigma_0(P')$.*

Proof. Let $P = \mathbb{D}[x(y).P_r \mid \overline{x}z.P_s]$ 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:

$$\begin{aligned}\sigma_0(P) = C^{\sigma}_{out}[D_1[\nu x, \mathit{chanx}.\mathit{chanx}\,\mathbf{m}\,- \mid x = \texttt{Channel}\ \mathit{chanx}\\ \mid D_2[\Leftarrow \mathbf{do}\ \{\mathit{checkx} \leftarrow \texttt{newMVar}\ ();\\ \texttt{putMVar}\ (\mathit{unchan}\ x)\ (z, \mathit{checkx});\\ \texttt{putMVar}\ \mathit{checkx}\ (); \tau(P_s)\}\\ \mid \Leftarrow \mathbf{do}\ \{(y, \mathit{checkx}) \leftarrow \texttt{takeMVar}\ (\mathit{unchan}\ x);\\ \texttt{takeMVar}\ \mathit{checkx}; \tau(P_r)\}]]]\end{aligned}$$

where D_1, D_2 are the σ-translations of $\mathbb{D}_1, \mathbb{D}_2$. Inspecting σ shows, that D_1, D_2 are $\mathbb{D}$-contexts. There is a sequence of $\xrightarrow{sr}$-reductions (see Fig. 11) such that

$$\begin{aligned}\sigma_0(P) \xrightarrow{sr,15} Q = \ & C^{\sigma}_{out}[D_1[\nu x, \mathit{chanx}.\mathit{chanx}\,\mathbf{m}\,- \mid x{=}\texttt{Channel}\ \mathit{chanx}\\ & \mid D_2[\Leftarrow \mathbf{do}\ \{\tau(P_s)\} \mid \Leftarrow \mathbf{do}\ \{\tau(P_r)[z/y]\}]]]\\ & \mid (\nu\,\mathit{checkx}.\mathit{checkx}\,\mathbf{m}\,())\end{aligned}$$

where $G = (\nu\,\mathit{checkx}.\mathit{checkx}\,\mathbf{m}\,())$ is garbage. Since

$$\begin{aligned}\sigma_0(P') = C^{\sigma}_{out}[D_1[\nu x, \mathit{chanx}.\mathit{chanx}\,\mathbf{m}\,- \mid x = \texttt{Channel}\ \mathit{chanx}\\ \mid D_2[\sigma(P_r[z/y]) \mid \sigma(P_s)]]],\end{aligned}$$

Lemma A.3 implies $Q \xrightarrow{sr,*} \sigma_0(P') \mid \nu\mathit{checkx}.\mathit{checkx}\,\mathbf{m}\,() \xrightarrow{gc} \sigma_0(P')$.

Lemma A.6. *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, $\sigma_0(P) \sim_c \sigma_0(P')$, and hence $\sigma_0(P) \sim_{c,0} \sigma_0(P')$,*

Proof. For rules (assocl) and (assocr), we have

$$\begin{aligned}\sigma_0(\mathbb{D}[(P_1 \mid P_2) \mid P_3]) = C[(\sigma(P_1) \mid \sigma(P_2)) \mid \sigma(P_3)] \equiv C[\sigma(P_1) \mid (\sigma(P_2) \mid \sigma(P_3))]\\ = \sigma_0(\mathbb{D}[P_1 \mid (P_2 \mid P_3)])\end{aligned}$$

for some *CH* $\mathbb{D}$-context C and thus, $\sigma_0(P) \equiv \sigma_0(P')$. For (nuup1), we have

$$\begin{aligned}&\sigma_0(\mathbb{D}[(\nu z.P_1) \mid P_2]) = C[\sigma(\nu z.P_1) \mid \sigma(P_2)])\\ &= C[\nu z, \mathit{chanz}.(\mathit{chanz}\,\mathbf{m}\,- \mid z = \texttt{Chan}\ \mathit{chanz} \mid \sigma(P_1)) \mid \sigma(P_2)]\\ &\equiv C[\nu z, \mathit{chanz}.(\mathit{chanz}\,\mathbf{m}\,- \mid z = \texttt{Chan}\ \mathit{chanz} \mid \sigma(P_1) \mid \sigma(P_2))]\\ &= \sigma_0(\mathbb{D}[\nu z.(P_1 \mid P_2)])\end{aligned}$$

for some *CH* $\mathbb{D}$-context C. Thus, $\sigma_0(P) \equiv \sigma_0(P')$. For (nuup2), we have

$$\begin{aligned}&\sigma_0(\mathbb{D}[\nu z.\nu x.P_1]) = C[\sigma(\nu z.\nu x.P_1)]\\ &= C[\nu x, \mathit{chanx}.\mathit{chanx}\,\mathbf{m}\,- \mid x = \texttt{Chan}\ \mathit{chanxx}\\ &\qquad\qquad \mid \nu z, \mathit{chanz}.(\mathit{chanz}\,\mathbf{m}\,- \mid z = \texttt{Chan}\ \mathit{chanz} \mid \sigma(P_1)))]\\ &\equiv \sigma_0(\mathbb{D}[\nu x.\nu z.P_1])\end{aligned}$$

$$
\begin{array}{ll}
\sigma_0(P) = C^{\sigma}_{out}[D_1[\nu x, \mathit{chanx}.\mathit{chanx}\,\mathsf{m}\,- \mid x{=}\texttt{Channel}\ \mathit{chanx} & \\
\quad \mid D_2[\Leftarrow \mathbf{do}\ \{\mathit{checkx} \leftarrow \texttt{newMVar}\ (); \texttt{putMVar}\ (\mathit{unchan}\ x)\ (z, \mathit{checkx}); & \\
\qquad\qquad \texttt{putMVar}\ \mathit{checkx}\ (); \tau(P_s)\} & \\
\quad \mid \Leftarrow \mathbf{do}\ \{(y, \mathit{checkx}) {\leftarrow} \texttt{takeMVar}\ (\mathit{unchan}\ x); \texttt{takeMVar}\ (\mathit{checkx}); \tau(P_r)\}]]] & \\
\xrightarrow{sr,nmvar}\xrightarrow{sr,beta} & C^{\sigma}_{out}[D_1[\nu x, \mathit{chanx}, \mathit{checkx}.\mathit{chanx}\,\mathsf{m}\,- \mid \mathit{checkx}\,\mathsf{m}\,() \mid x{=}\texttt{Channel}\ \mathit{chanx} \\
& \quad \mid D_2[\Leftarrow \mathbf{do}\ \{\texttt{putMVar}\ (\mathit{unchan}\ x)\ (z, \mathit{checkx}); \\
& \qquad\qquad \texttt{putMVar}\ \mathit{checkx}\ (); \tau(P_s)\} \\
& \qquad \mid \Leftarrow \mathbf{do}\ \{(y, \mathit{checkx}) {\leftarrow} \texttt{takeMVar}\ (\mathit{unchan}\ x); \\
& \qquad\qquad \texttt{takeMVar}\ (\mathit{checkx}); \tau(P_r)\}]]] \\
\xrightarrow{sr,cpce}\xrightarrow{sr,case} & C^{\sigma}_{out}[D_1[\nu x, \mathit{chanx}, \mathit{checkx}.\mathit{chanx}\,\mathsf{m}\,- \mid \mathit{checkx}\,\mathsf{m}\,() \mid x{=}\texttt{Channel}\ \mathit{chanx} \\
& \quad \mid D_2[\Leftarrow \mathbf{do}\ \{\texttt{putMVar}\ (\mathit{chanx})\ (z, \mathit{checkx}); \texttt{putMVar}\ \mathit{checkx}\ (); \tau(P_s)\} \\
& \qquad \mid \Leftarrow \mathbf{do}\ \{(y, \mathit{checkx}) {\leftarrow} \texttt{takeMVar}\ (\mathit{unchan}\ x); \\
& \qquad\qquad \texttt{takeMVar}\ (\mathit{checkx}); \tau(P_r)\}]]] \\
\xrightarrow{sr,pmvar}\xrightarrow{sr,beta} & C^{\sigma}_{out}[D_1[\nu x, \mathit{chanx}, \mathit{checkx}.\mathit{chanx}\,\mathsf{m}\,(z, \mathit{checkx}) \mid \mathit{checkx}\,\mathsf{m}\,() \\
& \quad \mid x{=}\texttt{Channel}\ \mathit{chanx} \mid D_2[\Leftarrow \mathbf{do}\ \{\texttt{putMVar}\ \mathit{checkx}\ (); \tau(P_s)\} \\
& \qquad\qquad \mid \Leftarrow \mathbf{do}\ \{(y, \mathit{checkx}) \leftarrow \texttt{takeMVar}\ (\mathit{unchan}\ x); \\
& \qquad\qquad\qquad \texttt{takeMVar}\ (\mathit{checkx}); \tau(P_r)\}]]] \\
\xrightarrow{sr,cpce}\xrightarrow{sr,case} & C^{\sigma}_{out}[D_1[\nu x, \mathit{chanx}, \mathit{checkx}.\mathit{chanx}\,\mathsf{m}\,(z, \mathit{checkx}) \mid \mathit{checkx}\,\mathsf{m}\,() \\
& \quad \mid x{=}\texttt{Channel}\ \mathit{chanx} \mid D_2[\Leftarrow \mathbf{do}\ \{\texttt{putMVar}\ \mathit{checkx}\ (); \tau(P_s)\} \\
& \qquad\qquad \mid \Leftarrow \mathbf{do}\ \{(y, \mathit{checkx}) {\leftarrow} \texttt{takeMVar}\ \mathit{chanx}; \\
& \qquad\qquad\qquad \texttt{takeMVar}\ (\mathit{checkx}); \tau(P_r)\}]]] \\
\xrightarrow{sr,tmvar}\xrightarrow{sr,beta} & \\
\quad \xrightarrow{sr,case} & C^{\sigma}_{out}[D_1[\nu x, \mathit{chanx}, \mathit{checkx}.\mathit{chanx}\,\mathsf{m}\,- \mid \mathit{checkx}\,\mathsf{m}\,() \mid x{=}\texttt{Channel}\ \mathit{chanx} \\
& \quad \mid D_2[\Leftarrow \mathbf{do}\ \{\texttt{putMVar}\ \mathit{checkx}\ (); \tau(P_s)\} \\
& \qquad \mid \Leftarrow \mathbf{do}\ \{\texttt{takeMVar}\ (\mathit{checkx}); \tau(P_r)[z/y]\}]]] \\
\xrightarrow{sr,tmvar}\xrightarrow{sr,beta} & C^{\sigma}_{out}[D_1[\nu x, \mathit{chanx}, \mathit{checkx}.\mathit{chanx}\,\mathsf{m}\,- \mid \mathit{checkx}\,\mathsf{m}\,- \mid x{=}\texttt{Channel}\ \mathit{chanx} \\
& \quad \mid D_2[\Leftarrow \mathbf{do}\ \{\texttt{putMVar}\ \mathit{checkx}\ (); \tau(P_s)\} \\
& \qquad \mid \Leftarrow \mathbf{do}\ \{\tau(P_r)[z/y]\}]]] \\
\xrightarrow{sr,pmvar}\xrightarrow{sr,beta} & C^{\sigma}_{out}[D_1[\nu x, \mathit{chanx}, \mathit{checkx}.\mathit{chanx}\,\mathsf{m}\,- \mid \mathit{checkx}\,\mathsf{m}\,() \mid x{=}\texttt{Channel}\ \mathit{chanx} \\
& \quad \mid D_2[\Leftarrow \mathbf{do}\ \{\tau(P_s)\} \mid \Leftarrow \mathbf{do}\ \{\tau(P_r)[z/y]\}]]] \\
\equiv & C^{\sigma}_{out}[D_1[\nu x, \mathit{chanx}.\mathit{chanx}\,\mathsf{m}\,- \mid x{=}\texttt{Channel}\ \mathit{chanx} \\
& \quad \mid D_2[\Leftarrow \mathbf{do}\ \{\tau(P_s)\} \mid \Leftarrow \mathbf{do}\ \{\tau(P_r)[z/y]\}]]] \mid (\nu \mathit{checkx}.\mathit{checkx}\,\mathsf{m}\,()) \\
= Q &
\end{array}
$$

Fig. 11. Reduction sequence $\sigma_0(P) \xrightarrow{sr,15} Q$ which is used in Lemma A.5

for some *CH* $\mathbb{D}$-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:

$$\begin{aligned}
\sigma_0(\mathbb{D}[!P]) &= C[\nu f.(\Leftarrow f \mid f = \mathbf{do}\ \{\texttt{forkIO}\ \tau(P); f\})] \\
&\xrightarrow{sr,cpce} C[\nu f.(\Leftarrow \mathbf{do}\ \{\texttt{forkIO}\ \tau(P); f\} \mid f = \mathbf{do}\ \{\texttt{forkIO}\ \tau(P); f\})] \\
&\xrightarrow{sr,fork} C[\nu f.(\Leftarrow \tau(P) \mid \Leftarrow \mathbf{do}\ \{\texttt{return ()}; f\} \mid f = \mathbf{do}\ \{\texttt{forkIO}\ \tau(P); f\})] \\
&\xrightarrow{sr,beta} C[\nu f.(\Leftarrow \tau(P) \mid \Leftarrow f \mid f = \mathbf{do}\ \{\texttt{forkIO}\ \tau(P); f\})] \\
&\equiv C[\Leftarrow \tau(P) \mid \nu f.(\Leftarrow f \mid f = \mathbf{do}\ \{\texttt{forkIO}\ \tau(P); f\})] \\
&\xrightarrow{sr,*} C[\sigma(P) \ \mid \nu f.(\Leftarrow f \mid f = \mathbf{do}\ \{\texttt{forkIO}\ \tau(P); f\})] = \sigma_0(\mathbb{D}[P \mid !P])
\end{aligned}$$

where $f \notin FV(P)$. This shows $\sigma_0(\mathbb{D}[!P]) \xrightarrow{sr,*} \sigma_0(\mathbb{D}[P \mid !P])$, using the previous items, and Lemma A.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 (see Proposition 3.6).

Lemma A.7. *Let P be a closed CH-process such that $P\ (\xrightarrow{sr,*} \cdot \sim_c)^*\ Q$. Then the following implications hold: i) If $Q{\downarrow}$, then $P{\downarrow}$ and ii) If $Q{\uparrow}$, then $P{\uparrow}$.*

Proof. We only show the first part. The proof for the second part can be obtained by replacing $\downarrow$ by $\uparrow$. The proof is by induction on the length of the relational sequence. Let $P\ (\xrightarrow{sr,*} \cdot \sim_c)^n\ \ P_1 \xrightarrow{sr,*} P_2 \sim_c Q$, where $Q{\downarrow}$. From $P_2 \sim_c Q$ we derive that also $P_2{\downarrow}$, and hence also $P_1{\downarrow}$, since $P_1 \xrightarrow{sr,*} P_2$.

Lemma A.8. *If a closed process P of the calculus $\Pi_{\texttt{Stop}}$ is successful, then $\sigma_0(P) \xrightarrow{sr,*} Q$ where Q is successful.*

Proof. Process P must contain `Stop` in a $\mathbb{D}$-context. Thus the translation generates a thread in a $\mathbb{D}$-context, that executes `takeMVar` *stop* and $Q \xrightarrow{sr,tmvar}\xrightarrow{sr,pmvar} D[\overset{\textsf{main}}{\Longleftarrow} \texttt{return}\ ()]$. 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 A.9. *Implication of may-convergence: Let P be a closed $\Pi_{\texttt{Stop}}$-process. If $P{\downarrow}$ then $\tau(P){\downarrow_0}$.*

Proof. Lemmas A.5 to A.8 imply that $\sigma_0(P) \xrightarrow{sr,*} Q$, where Q is successful, using induction on the length of a standard reduction of P, hence $\sigma(P){\downarrow_0}$. By Proposition A.4 this also shows $\tau_0(P){\downarrow}$ which is the same as $\tau(P){\downarrow_0}$.

Note that in the previous proof it is irrelevant whether we remove the garbage that stems from translated $\xrightarrow{dia}$-steps or not: In the former case we know that (gc) is a correct program transformation and thus does not change convergence, and in the latter case we carry the garbage with the whole reduction sequence.

$$\begin{tikzcd}
P \arrow[rrr, "\sigma_0"] \arrow[d, dashed, "{sr,*}"'] & & & \sigma_0(P) \arrow[d, "{sr,n}"'] \arrow[dl, dashed, "{(sr\cup\sim_c),*}"'] \\
P_1 \arrow[rr, dashed, "\sigma_0"] \arrow[d, dashed, "{sr,*}"'] & & \sigma_0(P_1) \arrow[dl, dashed, "{(sr\cup\sim_c),*}"'] \arrow[d, dashed, "{sr,n'}"', "n'<n"] & Q_1(succ.) \\
P_2 \arrow[r, dashed, "\sigma_0"] & \sigma_0(P_2)(succ.) & Q_2(succ.) &
\end{tikzcd}$$

Fig. 12. Proving reflection of may-convergence: structure of the induction step

A.3 The Translation Reflects May-Convergence

In this section we show that may-convergence of $\sigma_0(P)$ for a closed $\Pi_{\texttt{Stop}}$-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 $\Pi_{\texttt{Stop}}$-process P: $\tau_0(P)$ is closed; $\tau_0(P)$ contains the variable *stop* only in expressions of the form $\Leftarrow\texttt{takeMVar}\ \mathit{stop}$, and $\tau_0(P)$ is well-formed and well-typed.

Proposition A.10. *Let $P \in \Pi_{\texttt{Stop}}$ 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 $\mathit{Red} = \sigma_0(P) \xrightarrow{sr,n} Q_1$.

$$\begin{tikzcd}
P \arrow[rr, "\sigma_0"] \arrow[d, dashed, "{sr,*}"'] & & \sigma_0(P) \arrow[d, "{sr,n}"] \arrow[dl, dashed, "{(sr\cup\sim_c),*}"] \\
P_1 \arrow[r, dashed, "\sigma_0"] & \sigma_0(P_1)(succ.) & Q_1(succ.)
\end{tikzcd}$$

In this diagram and in the diagram in Fig. 12 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 $C^{\sigma}_{out}[\mathbb{D}[\Leftarrow\texttt{takeMVar}\ \mathit{stop}]]$, and it standard-reduces in two steps to a successful process. In this case, the only possible reason is that P contains $\texttt{Stop}$ 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 shown in Fig. 12. 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 *CH*-standard reduction sequence, extend it with missing reduction steps, which do not change success, and construct a corresponding standard reduction sequence 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 $S; Red'$ 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_1 is reached. In the first case, everything is fine, and a process P_1 can be determined. In the second case, the subsequence has to be removed, or 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, and that a (gc)-step has to be inserted as a $\sim_c$-step.

Referring to Definition A.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 (newmvar)-, (tmvar)-, and (pmvar)-reductions, the reduction sequence starts as follows

(ia-0) $\mathit{checkx} \leftarrow$ `newMVar`();
(ia-1) `putMVar` $(\mathit{unchan}\ x)\ (y, \mathit{checkx})$ in thread a;
(ia-2) `takeMVar` $(\mathit{unchan}\ x)$ in thread b $(\neq a)$;
(ia-3) `takeMVar` checkx in thread b;
(ia-4) `putMVar` checkx () in thread a.

However, there may be deviations, like interleaving of reduction steps in parallel threads, or an incomplete subsequence. We analyse the possibilities in a reduction sequence, where we omit the non-interfering interleaved actions. Of course, we assume that the programs have the commands for (ia-0);(ia-1);(ia-4) as a sequence in one thread-program and the commands for (ia-2);(ia-3) as a sequence in the other thread-program.

1. (ia-0) is the only action for *checkx*; no further action; thread a is blocked.
2. (ia-0);(ia-1) is performed in thread a; no other thread uses the MVar $(\mathit{unchan}\ x)$; thread a is blocked.
3. (ia-0);(ia-1);(ia-2) is performed first in thread a, and some other thread b won the race for (ia-2), and then no further progress; threads a, b are blocked. `takeMVar` *checkx* is not possible in another thread, since the name is unknown to other threads. However, the MVar $(\mathit{unchan}\ x)$ is already available for other threads.

4. (ia-0);(ia-1);(ia-2);(ia-3) is performed with (ia-0);(ia-1) in thread a, and then (ia-2); (ia-3) in thread b; thread a is blocked, but thread b can proceed. MVar *checkx* is not garbage.
5. (ia-0);(ia-1);(ia-2);(ia-3);(ia-4) is performed with (ia-0);(ia-1);(ia-4) in thread a, and (ia-2); (ia-3) in thread b. Thread a is released after the full sequence, thread b after (ia-3). Moreover, after (ia-4), the MVar *checkx* can no longer be used and is garbage.

We show that modifying the reduction sequence *Red*, keeping the sr-property, results in another reduction sequence *Red′* that starts with the ideal 5 reductions, followed by *Red″* also ends in a successful process, such that *Red″* contains less standard-reduction steps.

Now we focus the ia-related reduction steps in *Red*. We remove the following subsequences: In case there is a thread such that after an (ia-0) or after an (ia-0);(ia-1) step, then the thread is blocked; Then we remove these reduction steps from *Red*. We do this in all cases.

Now we focus the earliest occurrence in *Red* of an (ia-2)-related reduction step. If the (ia)-subsequence corresponding to the first one is complete, then we move it to the start of the sequence. Of course the non-MVar related sr-steps in the corresponding threads a, b are also moved. In the following we do not mention these sr-reductions.
Now let us consider the incomplete ones:

- If exactly (ia-0);(ia-1);(ia-2) are in *Red*, then thread a, b do not further contribute to the success, however, MVar (*unchan x*) may be used for the success (triggered by another thread). We add the reductions (ia-3) and (ia-4) in order to complete the ia-sequence, which makes the reduction sequence longer, but does not add (ia-2)-reduction steps.
- If exactly (ia-0);(ia-1);(ia-2);(ia-3) are in *Red*, then we add the missing (ia-4) step without changing success and the rest of the reduction sequence.

In order to apply the induction hypothesis, we insert a (gc)-step to remove the check-MVar that was only used for one interaction step in the translation. This (gc) is correct, hence we can represent it by a $\sim_c$-step.

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)-reduction. 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 A.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 A.6, 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 $\Leftarrow\tau(P_r)$ 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 $\Pi_{\mathtt{Stop}}$ to a successful process.

Propositions A.9 and A.10 imply:

Proposition 4.5. *Let $P \in \Pi_{\mathtt{Stop}}$ be closed. Then τ_0 is convergence-equivalent for $\downarrow$, i.e. $P\downarrow$ is equivalent to $\tau_0(P)\downarrow$. This also implies that $P\Uparrow$ is equivalent to $\tau(P)\Uparrow$.*

A.4 Equivalence of Should-Convergence of the Translation

We argue that the translation τ is invariant w.r.t. should-convergence, where we work with may-divergence and where Proposition 4.5 is very useful since it is the induction base.

Proposition A.11 (Preservation of May-Divergence). *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 $\sigma_0(P) \xrightarrow{sr,*} Q$ with $Q\Uparrow$.*

Proof. We use induction on n to show that $P \xrightarrow{sr,n} P_1$ where $P_1\Uparrow$ implies $\sigma_0(P)(\xrightarrow{sr,*} \cdot \sim_c)^* Q'$ where Q' is must-divergent. The base case $n = 0$ is covered by Proposition 4.5. For the induction step, Lemmas A.5 and A.6 can be applied for the first reduction step and then the induction hypothesis can be applied to show the claim. Finally, we apply item ii) of Lemma A.7 to derive the claim.

The last case is to show that reflection of may-divergence holds, i.e. $\sigma_0(P)\uparrow \implies P\uparrow$. This is almost similar to the arguments for $\sigma_0(P)\downarrow \implies P\downarrow$ 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 A.12. *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 $P \xrightarrow{sr,*} P_1$ with $P_1\Uparrow$, and another standard reduction sequence $\sigma_0(P) \xrightarrow{sr,n} Q_1'$ such that $Q_1'\Uparrow$ and $Q_1' = \sigma_0(P_1')$.*

Proof. The following diagram gives an orientation on the goals of the proof and the induction on n.

$$\begin{tikzcd}
P \arrow[rr, "\sigma_0"] \arrow[d, dashed, "{sr,*}"'] & & \sigma_0(P) \arrow[d, "{sr,n}"] \arrow[dl, dashed, "{(sr\cup\sim_c),*}"] \\
P_1\Uparrow \arrow[r, dashed, "\sigma_0"] & \sigma_0(P_1)\Uparrow & Q_1\Uparrow
\end{tikzcd}$$

Most arguments for a construction of the the reduction sequence $\sigma_0(P) \xrightarrow{sr\cup\sim_c,*} \sigma_0(P_1)$ are already in the proof of Proposition A.10.
However, the treatment of the incomplete (ia)-reduction sequences is different, so we make it explicit.
The cases are:

1. In the case that only the (ia-0) is present, we can omit it, since it is correct.
2. If at least (ia-0);(ia-1);(ia-2) is there, then we add the missing reductions. The argument is: If $\sigma_0(P) \xrightarrow{sr,*} Q{\Uparrow}$ and $Q \xrightarrow{sr\cup\sim_c,*} Q'$ then also $Q'{\Uparrow}$, and we can perform the extra reductions (like (ia-3);(ia-4) from Q, and then shift them closer to the (ia-0);(ia-1);(ia-2)-sequence. This makes the ia-sequence complete, and does not change the induction measure.
3. The case that the incomplete sequence is (ia-0);(ia-1), and (ia-2) is not present, needs more arguments: There are two possibilities:
   (a) There is a reduction sequence starting from Q, which contains the needed (ia-2). Then we will shift first the (ia-2) such that $Q \xrightarrow{\text{(ia-2)}} Q'$, where obviously $Q'{\Uparrow}$. Then we shift the (ia-2) further to the left, such that we have the sequence (ia-0);(ia-1);(ia-2), and we can then use the other cases. Here we have modify the induction measure: first the number of (ia-2) in the reduction sequence to Q, and as second component the total number of (ia)-reductions in the sequence $\sigma_0(P) \xrightarrow{sr,*} Q$. We have removed more than one (ia)-reduction in the induction, hence we can apply the induction hypothesis.
   (b) There is no reduction sequence starting from Q with the needed (ia-2). In this case the argument uses that the current CH-program is a translation of a $\Pi_{\texttt{Stop}}$-process. We remove the (ia-1) from the reduction sequence. The reduction leads to Q_1, which is the result of the reduction sequence leading to Q, but without the (ia-1)-reduction. Now there may be another thread that can access the MVar (*unchan* x), again using (ia-0);(ia.1) in the other thread. However, since from Q there is no (ia-2) possible for MVar (*unchan* x), the same must hold for Q_1. Hence, the removal permits only small local changes, but there is still no successful reduction from Q_1. Hence $Q_1{\Uparrow}$. Of course, we can now also remove the (ia-0)-reduction.

   Note that we also have to use the (gc)-reduction removing the check-MVar in case of a complete (ia)reduction in CH, which adds a $\sim_c$ to the combined reduction sequence.

Using these arguments the proof is otherwise completely analogous to the proof of Proposition A.10, and we are done.

Proposition 4.6. *Let $P \in \Pi_{\texttt{Stop}}$ be closed. Then τ_0 is convergence-equivalent for $\Downarrow$, i.e. $P{\Downarrow}$ is equivalent to $\tau_0(P){\Downarrow}$.*

B Proofs and Details on Translations with Global Names

B.1 Failing Translations with Less than Three Check-MVars and the Interprocess Check Restriction: Examples

Before we prove that the first translation of Table 2 is invariant for may- and should-convergence, we show, how similar translations with less check-MVars fail. The reason is usually that the translated sequences for send and receive are not forced to be completely executed. For example, the translation of $\overline{x}y \mid x(y).\overline{x}y.\texttt{Stop}$ with one check-MVar where the sender program is $[\texttt{putS}, \texttt{takeC}]$; and the receiver program is: $[\texttt{putC}, \texttt{takeS}]$ confuses the sequence for the program consisting of two threads, and where the π-process is not may-convergent, but the translated program may-converges. The program is written in the first line as two threads, and the sequence of execution in the second line:

$$\begin{array}{cc|cccccc}
[\texttt{putS}, & \texttt{takeC}] & [\texttt{putC}, & \texttt{takeS}, & \texttt{putS}, & \texttt{takeC}, & \texttt{Stop}] \\
1 & - & 2 & 3 & 4 & 5 & 6
\end{array}$$

The situation is similar for all other translations using one or two check-MVars, and where every check-MVar is written and read only once per translated interact. Either the communication is not synchronous, or the sequences of send and receive that do not belong to the intended interact can be interleaved.

For documentation purposes, we detail an example for a bad behavior for a translation using two check-MVars and which is similar to the first one, where this example is a result of using our simulator. The translation is

send	*receive*
$[\texttt{putS}, \texttt{putC}^1, \texttt{takeC}^2]$	$[\texttt{takeC}^1, \texttt{putC}^2, \texttt{takeS}]$

The process in π-calculus notation is $\overline{x}z.\overline{z}a.\texttt{Stop} \mid \overline{x}w.\overline{w}a.\texttt{Stop} \mid x(y).y(u)$, which is should-convergent. We write the variable name of the value-MVar also at the check-MVar commands. Note that after a $\texttt{takeC}_x\ y$, the y in the rest of the thread is changed accordingly.
The program consists of three threads, where the sequence of the problematic execution is:

$$\begin{array}{ccccccc}
[\texttt{putS}_x\ z, & \texttt{putC}^1_x, & \texttt{takeC}^2_x, & \texttt{putS}_z\ a, & \texttt{putC}^1_z, & \texttt{takeC}^2_z, & \texttt{Stop}] \\
1 & 2 & & & & & \\
[\texttt{putS}_x\ w, & \texttt{putC}^1_x, & \texttt{takeC}^2_x, & \texttt{putS}_w\ a, & \texttt{putC}^1_w, & \texttt{takeC}^2_w, & \texttt{Stop}] \\
6 & 7 & 8 & 9 & 10 & & \\
[\texttt{takeC}^1_x, & \texttt{putC}^1_x, & \texttt{takeS}_x\ y, & \texttt{takeC}^1_y, & \texttt{putC}^2_y, & \texttt{takeS}_y\ u] & \\
3 & 4 & 5 & & & &
\end{array}$$

The final program is deadlocked but not successful, hence this translation is not invariant for should-convergence.

$$\begin{aligned}
\psi_{0,T}(P) &= \nu stop.(\overset{\mathsf{main}}{\Longleftarrow} \texttt{putMVar}\ stop\ ()\ \mid\ stop\,\mathbf{m}\,()\ \mid\ \psi_T(P))\\
\psi_T(\nu x.P) &= \nu x, contx, checkx_1, \ldots, checkx_n.\\
&\qquad\qquad (contx\,\mathbf{m}\,-\mid checkx_1\,\mathbf{m}\,-\mid \ldots \mid checkx_n\,\mathbf{m}\,-\\
&\qquad\qquad\ \mid x = \texttt{Channel}\ contx\ \ checkx_1 \ldots checkx_n \mid \psi_T(P))\\
\psi_T(\overline{x}y.P) &= \Leftarrow \phi_T(\overline{x}y.P)\\
\psi_T(x(y).P) &= \Leftarrow \phi_T(x(y).P)\\
\psi_T(0) &= \Leftarrow \texttt{return ()}\\
\psi_T(\texttt{Stop}) &= \Leftarrow \texttt{takeMVar}\ stop\\
\psi_T(P \mid Q) &= \psi_T(P) \mid \psi_T(Q)\\
\psi_T(!P) &= \nu f.(\Leftarrow f \mid f = \mathbf{do}\ \{\texttt{forkIO}\ \phi_T(P); f\})
\end{aligned}$$

Fig. 13. Induced translations ψ_T and $\psi_{0,T}$ for a gstb-translation T using n check-MVars

B.2 Correctness of Translations with Global Names

In the correctness proof of translation τ and τ_0, we defined the top-down translation σ, σ_0. For showing correctness of induced translations ϕ_T, we follow a similar way, and define induced top-down translations ψ_T and $\psi_{0,T}$ in Fig. 13, for a given gstb-translation $T = (T_{send}, T_{receive})$ using n check-MVars.

We first show the strong relation between ϕ_T and ψ_T:

Proposition B.1. *For all $P \in \Pi_{\texttt{Stop}}$ i) $\phi_{0,T}(P) \xrightarrow{sr,*} \psi_{0,T}(P)$, ii) $\Leftarrow \phi_T(P) \xrightarrow{sr,*} \psi_T(P)$, iii) $\psi_T(P) \sim_c \Leftarrow \phi_T(P)$ and iv) $\psi_{0,T}(P) \sim_c \phi_{0,T}(P)$.*

Proof. Parts i) and ii) can be proved similar to Lemma A.3. Part i) is completely analogous, while for part ii) the translation of the ν-operator is different, and thus the construction in the induction for this case is

$$\begin{aligned}
\Leftarrow \phi_T(\nu x.P)(\xrightarrow{sr,nmvar}\xrightarrow{sr,beta}\xrightarrow{sr,tmvar}\xrightarrow{sr,beta})^{n+1} &\xrightarrow{sr,mkbinds}\\
\nu x, contx, checkx_1, \ldots, checkx_n&.\\
(\Leftarrow \phi_T(P) \mid contx\,\mathbf{m}\,- \mid checkx_1\,\mathbf{m}\,- &\mid \ldots \mid checkx_n\,\mathbf{m}\,-\\
&\mid x = \texttt{Channel}\, contx\ \ checkx_1 \ldots checkx_n)
\end{aligned}$$

where n is the number of check-MVars and where all $(sr, tmvar)$-steps are deterministic. Parts iii) and iv) follow from Proposition 3.6 analogously to the proof of Proposition A.4.

Now, we can prove preservation of may-convergence for any induced translation of an executable gstb-translation, and preservation of may-divergence, provided that must-divergence equivalence already holds:

Proposition B.2. *Let T be an executable gstb-translation. Let P be a closed $\Pi_{\texttt{Stop}}$-process.*

- *If $P\downarrow$ then $\phi_{0,T}(P)\downarrow$.*
- *If for all closed $Q \in \Pi_{\texttt{Stop}}$: $Q\Uparrow \iff \psi_{0,T}(Q)\Uparrow$ then the implication $P\uparrow \implies \phi_{0,T}(P)\uparrow$ holds.*

Proof. The proof of the first item is analogous to the proof of Proposition A.9 by an induction on the given converging reduction sequence for P, that constructs a converging sequence for $\psi_{0,T}(P)$ and then applies Proposition B.1 to derive the sequence for $\phi_{0,T}(P)$. The second item follows also by an induction on the given reduction sequence for P to a must-divergent process, where the base case is covered by the precondition.

We sketch the main steps, required to perform the induction:

- For the base case, one can verify that if a closed process P of the calculus $\Pi_{\texttt{Stop}}$ is successful, then $\psi_{0,T}(P) \xrightarrow{sr,*} Q$ where Q is successful.
- For the translation of $\xrightarrow{dia}$-steps the following claim hold:

  Let $P \in \Pi_{\texttt{Stop}}$ be a closed process with $P \xrightarrow{dia} P'$. Then there is a sequence $\psi_{0,T}(P) \xrightarrow{sr,*} \psi_{0,T}(P')$ in the *CH*-calculus.

  The proof is similar to the proof of Lemma A.5 where one can verify that
  - the sr-reduction for the *CH*-program belonging to `putS` or `putC` consists of $\xrightarrow{sr,cpcx}\xrightarrow{sr,case}\xrightarrow{sr,pmvar}\xrightarrow{beta}$
  - the sr-reduction for the *CH*-program belonging to `takeS` or `takeC` consists of $\xrightarrow{sr,cpcx}\xrightarrow{sr,case}\xrightarrow{sr,takeMVar}\xrightarrow{beta}$

  and that no garbage (in difference to Lemma A.5 is generated, and that after executing sender and receiver program the requirements on an executable gstb-translation imply that the corresponding MVars are again empty.
- For other reduction steps in the $\Pi_{\texttt{Stop}}$-calculus the following claim can be proved analogous to Lemma A.6 with a different translation of νx, but this does not change the result:

  Let $P, P' \in \Pi_{\texttt{Stop}}$ be closed, such that $P \xrightarrow{dsc} P'$. Then $\psi_{0,T}(P) \xrightarrow{sr,*} \psi_{0,T}(P')$, if rule (replunfold) is used, and $\psi_{0,T}(P) \equiv \psi_T(P')$, otherwise. In particular, $\psi_{0,T}(P) \sim_c \psi_{0,T}(P')$.

To prove convergence equivalence for $\phi_{0,T}$ for a given gstb-translation it thus suffices to show, that $\psi_{0,T}$ reflects may-convergence and may-divergence

B.3 Correctness of a Translation with the Interprocess Check Restriction and Using Three Global Check-MVars

We sketch that the first translation in Table 2 is correct. The translation is

$$\mathfrak{T}_1 = (T_{send}, T_{receive}) = ([\texttt{putS}, \texttt{putC}^1, \texttt{takeC}^2, \texttt{putC}^3], [\texttt{takeC}^1, \texttt{putC}^2, \texttt{takeC}^3, \texttt{takeS}])$$

We conjecture that all four translations are correct using similar arguments. We leave open whether the translations 5,6 of Table 2 are correct. In the proof we mention only the reduction steps that correspond to actions.

Theorem B.3. *For all closed π-processes P and the translation $\phi_{0,\mathfrak{T}_1}$, the equivalences $P{\downarrow} \iff \phi_{0,\mathfrak{T}_1}(P){\downarrow}$ and $P{\Downarrow} \iff \phi_{0,\mathfrak{T}_1}(P){\Downarrow}$ hold.*

Proof. 1. If $P{\downarrow}$, then $\phi_{0,\mathfrak{T}_1}(P){\downarrow}$ follows from Proposition B.2.

2. Let us assume that $\psi_{0,\mathfrak{T}_1}(P)\downarrow$, and let $\psi_{0,\mathfrak{T}_1}(P) \xrightarrow{*} s_0$ be a successful reduction sequence. The task is now to find a rearrangement of the reduction sequence which remains successful, and that can be translated back into a $\Pi_{\mathtt{Stop}}$-calculus reduction sequence. We only consider the MVar accesses ignoring the other reduction steps and try to make an appropriate rearrangement. We can assume that there is an access to an MVar, say x, in the reduction sequence. Due to the initialization, the first one is a $\mathtt{putS}_x$, i.e. a `putMVar` on x.

   If the sequence $\mathtt{putC}_x^1, \mathtt{takeC}_x^2, \mathtt{putC}_x^3$in the thread of $\mathtt{putS}_x$ is not finished, then this thread does not contribute to the success. Also the $T_{receive}$ sequence cannot be finished. Hence we can remove all the reductions from both MVar-access-sequences without changing the success. This makes the reduction sequence smaller and we can use induction. If the T_{send}-sequence is complete, but the $\mathtt{takeS}_x$ of the $T_{receive}$-sequence is missing, then it is impossible that another $T_{receive}$-sequence starts (see Proposition 5.8) Hence we can add the $\mathtt{takeC}_x^3, \mathtt{takeS}_x$-sequence or only the action $\mathtt{takeS}_x$ without changing success and thus the two sequences are completely done, and retranslatable. We can move the reductions as far to the left as possible, close to the start of T_{send} with $\mathtt{putS}_x$. Then we can apply induction on the remaining part of the reduction sequence.

   Thus we have $\psi_{0,\mathfrak{T}_1}(P)\downarrow \implies P\downarrow$. By Proposition B.1 this also shows $\phi_{0,\mathfrak{T}_1}(P)\downarrow \implies P\downarrow$.
3. Now we can assume that $P\downarrow \iff \psi_{0,\mathfrak{T}_1}(P)\downarrow$, and also $P\Uparrow \iff \psi_{0,\mathfrak{T}_1}(P)\Uparrow$.
4. The implication $P\uparrow \implies \phi_{0,\mathfrak{T}_1}(P)\uparrow$ now follows from Proposition B.2.
5. The implication $\phi_{0,\mathfrak{T}_1}(P)\uparrow \implies P\uparrow$ is a bit more complex and requires again rearranging the reduction sequence. We show $\psi_{0,\mathfrak{T}_1}(P)\uparrow \implies P\uparrow$ and apply Proposition B.2. to derive the result for $\phi_{0,\mathfrak{T}_1}(P)\uparrow$.

   However, we now have to argue more in the case of reduction subsequences that do not correspond to a translated interact. Therefore we have to show that the removal of certain reduction steps and/or the addition of reduction steps is invariant w.r.t. must-divergent processes.

   Due to Proposition 5.8, it is sufficient to consider a reduction sequence that exhibits may-divergence, and look for the leftmost single MVar accesses:

   Let $\psi_{0,\mathfrak{T}_1}(P) \xrightarrow{sr,*} s_\infty$ be a reduction sequence where s_∞ is must-divergent.
   - Let there be a leftmost $\mathtt{putS}_x$ for channel x, and also perhaps a subsequent $\mathtt{putC}_x^1$, but no more actions for x in the sequence. Let s_2 be the process after performing $\mathtt{putS}_x$ (or $\mathtt{putC}_x^1$, if present), and s_1 be the process before performing $\mathtt{putS}_x$ Then we distinguish two cases: i) If there is no other thread which can perform $\mathtt{takeC}^1$, then we can remove the operations, constructing a s'_∞ which still must be must-divergent: Any other send-sequence on x starting with $\mathtt{putS}_x$ performed by another thread also cannot be completed, since there is no receiver that can perform $\mathtt{takeC}^1$ ii) There is another thread which can perform $\mathtt{takeC}^1$ (but does not do it). Then this thread is blocked and still present in s_∞. Thus we can add the step after s_∞ and derive an s'_∞ and then we shift the reductions that perform the $\mathtt{takeC}^1$ to the left.

- The next case is the execution of $\mathtt{putS}_x$ and $\mathtt{putC}^1_x$ in thread 1, and $\mathtt{takeC}^1_x$ in thread 2, but no further actions in the reduction sequence. Now it is not possible to argue using removal of the steps, since there may be a may-convergent reduction sequence using another thread for $\mathtt{takeC}^1_x$. Now Proposition 5.8, can be used. If we look for a possible reduction sequence from s_∞, then there is one reduction sequence (almost sr-reduction sequence if we ignore the main thread) that executes the complete interact for x. Since s_∞ is must-divergent, we can add reduction steps and the result will be must-divergent. Hence we can prolong the reduction sequence and also by shifting the missing reduction to the left obtain a sequence where the translated interact is complete.
- the final case is that we have the 8 actions for channel x in two threads. These can be shifted to the left, since there is no disturbance with other threads, and we obtain a complete interact that is retranslatable.

B.4 Correctness of a Translation without the Interprocess Check Restriction and Two Global Check-MVars

We sketch that the translation $\mathfrak{T}_7$ in Table 3 is correct. The translation is

$$\mathfrak{T}_7 = (T_{send}, T_{receive}) = ([\mathtt{putC}^1, \mathtt{putS}, \mathtt{takeC}^2, \mathtt{takeC}^1], [\mathtt{takeS}, \mathtt{putC}^2])$$

We conjecture that the second translation in Table 3 can be proved correct using similar arguments. We mention only the reductions that correspond to actions.

Theorem B.4. *For all π-processes P and the translation $\phi_{0,\mathfrak{T}_7}$, the equivalences $P{\downarrow} \iff \phi_{0,\mathfrak{T}_7}(P){\downarrow}$ and $P{\Downarrow} \iff \phi_{0,\mathfrak{T}_7}(P){\Downarrow}$ hold.*

Proof. We can reason similar to the proof of Theorem B.3. Concentrating on the core of the proof, it suffices to show that $\psi_{\mathfrak{T}_7}(P){\downarrow} \implies P{\downarrow}$ and $\psi_{\mathfrak{T}_7}(P){\uparrow} \implies P{\uparrow}$, where for the second implication we can assume that $Q{\Uparrow} \iff \psi_{\mathfrak{T}_7}(Q){\Uparrow}$ holds for all closed $\Pi_{\mathtt{Stop}}$-processes Q.

1. Let us assume that $\psi_{\mathfrak{T}_7}(P)$, and let $\psi_{\mathfrak{T}_7}(P) \xrightarrow{*} s_0$ be a successful reduction sequence. The task is now to find a rearrangement of the reduction sequence which remains successful, and that can be translated back into a $\Pi_{\mathtt{Stop}}$-calculus reduction sequence. We only consider the MVar accesses ignoring the other reduction steps and try to make an appropriate rearrangement. We can assume that there is an access to an MVar, say x, in the reduction sequence. Due to the initialization, the first one is a $\mathtt{putC}^1_x$. If the sequence $\mathtt{putS}_x, \mathtt{takeC}^2_x, \mathtt{takeC}^1_x$ in the thread of $\mathtt{putC}^1_x$ is not finished, then this thread does not contribute to the success. Also the $T_{receive}$ sequence cannot be finished. Hence we can remove all the reductions from both MVar-access-sequences without changing the success. This makes the reduction sequence smaller and we can use induction. If the T_{send}-sequence is complete, but the $\mathtt{takeS}_x$ of the $T_{receive}$-sequence is missing, then it is impossible that another $T_{receive}$-sequence starts (see Proposition 5.10) Hence

we can add the $\texttt{takeS}_x, \texttt{putC}^2_x$-sequence without changing success and thus the two sequences are completely done, and retranslatable. We can move the reductions as far to the left as possible, close to the start of T_{send} with $\texttt{putC}^1_x$. Then we apply induction on the remaining part of the reduction sequence.

2. The implication $\psi_{\mathfrak{T}_7}(P)\uparrow \implies P\uparrow$ requires again rearranging the reduction sequence. However, we now have to argue more in the case of reduction subsequences that do not correspond to a translated interact. Therefore we have to show that the removal of certain reduction steps and/or the addition of reduction steps is invariant w.r.t. must-divergent processes. Due to Proposition 5.10, it is sufficient to consider a reduction sequence that exhibits may-divergence, and look for the leftmost single MVar accesses: Let $\psi_{\mathfrak{T}_7}(P) \xrightarrow{sr,*} s_\infty$ be a reduction sequence where s_∞ is must-divergent.

   - Let there be a leftmost $\texttt{putC}^1_x$ for channel x, and also perhaps a subsequent $\texttt{putS}_x$, but no more actions for x in the sequence. If there is no thread that could perform $\texttt{takeS}_x$, then we can remove the $\texttt{putC}^1_x$. and $\texttt{putS}_x$-executions and the adapted process s'_∞ remains to be must-divergent, since any other sender that performs $\texttt{putC}^1_x$ also cannot be completed. If there is a thread that can perform $\texttt{takeS}_x$, then this is still possible in s_∞. Thus we add the $\texttt{takeS}_x$ (and the $\texttt{putS}_x$ if needed) for s_∞ and then shift the corresponding reductions to the left. Then we can use another case of this proof.
   - The next case is the execution of $\texttt{putC}^1_x$ and $\texttt{putS}_x$ in thread 1, and $\texttt{takeS}_x$ in thread 2, but no further actions in the reduction sequence. Now Proposition 5.10 implies that if we look for a possible reduction sequence from s_∞, then there is one reduction sequence (almost sr-reduction sequence if we ignore the main thread) that executes the complete interact for x. Since s_∞ is must-divergent, we can add reduction steps and the result will be must-divergent. Hence we can prolong the reduction sequence and also by shifting the missing reduction to the left obtain a sequence where the translated interact is complete.
   - The next case is the execution of $\texttt{putC}^1_x$ and $\texttt{putS}_x$ (and perhaps $\texttt{takeC}^2_x$) in thread 1, and $\texttt{takeS}_x, \texttt{putC}^2_x$ in thread 2, but no further actions in the reduction sequence. Again Proposition 5.10 implies that if we look for a possible reduction sequence from s_∞, then there is one reduction sequence (almost sr-reduction sequence if we ignore the main thread) that executes the complete interact for x. Since s_∞ is must-divergent, we can add reduction steps and the result will be must-divergent. Hence we can prolong the reduction sequence and also by shifting the missing reduction to the left obtain a sequence where the translated interact is complete.
   - the final case is that we have the 6 actions for channel x in two threads. These can be shifted to the left, since there is no disturbance with other threads, and we obtain a complete interact that is retranslatable.

$$\frac{\Gamma \vdash e :: \texttt{IO ()}}{\Gamma \vdash \Leftarrow e :: \texttt{wt}} \quad \frac{\Gamma \vdash e :: \mathsf{t}}{\Gamma \vdash x = e :: \texttt{wt}} \quad \frac{\Gamma \vdash P_1 :: \texttt{wt},\ \Gamma \vdash P_2 :: \texttt{wt}}{\Gamma \vdash P_1 \mid P_2 :: \texttt{wt}} \quad \frac{\Gamma(x) = \texttt{MVar}\ \mathsf{t}, \Gamma \vdash e :: \mathsf{t}}{\Gamma \vdash x\,\mathbf{m}\,e :: \texttt{wt}}$$

$$\frac{\Gamma(x) = \texttt{MVar}\ \mathsf{t}}{\Gamma \vdash x\,\mathbf{m} - :: \texttt{wt}} \quad \frac{\Gamma \vdash P :: \texttt{wt}}{\Gamma \vdash \nu x.P :: \texttt{wt}} \quad \frac{\Gamma \vdash e :: \mathsf{t}}{\Gamma \vdash \texttt{return}\ e :: \texttt{IO}\ \mathsf{t}} \quad \frac{\Gamma \vdash e_1 :: \texttt{IO}\,\mathsf{t}_1, \Gamma \vdash e_2 :: \mathsf{t}_1 \to \texttt{IO}\,\mathsf{t}_2}{\Gamma \vdash e_1 \mathbin{\texttt{>>=}} e_2 :: \texttt{IO}\,\mathsf{t}_2}$$

$$\frac{\Gamma \vdash e :: \texttt{IO ()}}{\Gamma \vdash \texttt{forkIO}\ e :: \texttt{IO ()}} \quad \frac{\Gamma \vdash e :: \texttt{MVar}\ \mathsf{t}}{\Gamma \vdash \texttt{takeMVar}\ e :: \texttt{IO}\ \mathsf{t}} \quad \frac{\Gamma \vdash e_1 :: \texttt{MVar}\ \mathsf{t},\ \Gamma \vdash e_2 :: \mathsf{t}}{\Gamma \vdash \texttt{putMVar}\ e_1\ e_2 :: \texttt{IO ()}}$$

$$\frac{\Gamma \vdash e :: \mathsf{t}}{\Gamma \vdash \texttt{newMVar}\ e :: \texttt{IO (MVar}\ \mathsf{t}\texttt{)}} \quad \frac{\forall i : \Gamma \vdash e_i :: \mathsf{t}_i,\ \mathsf{t}_1 \to \ldots \to \mathsf{t}_n \to \mathsf{t}_{n+1} \in \texttt{types(c)}}{\Gamma \vdash (c\ e_1\ \ldots\ e_{\mathrm{ar}(c)}) :: \mathsf{t}_{n+1}}$$

$$\frac{\forall i : \Gamma(x_i) = \mathsf{t}_i,\ \forall i : \Gamma \vdash e_i :: \mathsf{t}_i,\ \Gamma \vdash e :: \mathsf{t}}{\Gamma \vdash (\texttt{letrec}\ x_1 = e_1,\ \ldots\ x_n = e_n\ \texttt{in}\ e) :: \mathsf{t}} \quad \frac{\Gamma \vdash e_1 :: \mathsf{t}_1 \to \mathsf{t}_2,\ \Gamma \vdash e_2 :: \mathsf{t}_1}{\Gamma \vdash (e_1\ e_2) :: \mathsf{t}_2}$$

$$\frac{\Gamma(x) = \mathsf{t}_1,\ \Gamma \vdash e :: \mathsf{t}_2}{\Gamma \vdash (\lambda x.e) :: \mathsf{t}_1 \to \mathsf{t}_2} \quad \frac{\Gamma(x) = \mathsf{t}}{\Gamma \vdash x :: \mathsf{t}} \quad \frac{\begin{array}{c}\Gamma \vdash e_1 :: \mathsf{t}_1,\ \Gamma \vdash e_2 :: \mathsf{t}_2 \\ \mathsf{t}_1 = \mathsf{t}_3 \to \mathsf{t}_4 \text{ or } \mathsf{t}_1 = (T \ldots)\end{array}}{\Gamma \vdash (\texttt{seq}\ e_1\ e_2) :: \mathsf{t}_2}$$

$$\frac{\Gamma \vdash e :: \mathsf{t}_1 \text{ and } \mathsf{t}_1 = (T\ \ldots),\ \forall i : \Gamma \vdash (c_i\ x_{1,i}\ \ldots\ x_{n_i,i}) :: \mathsf{t}_1,\ \forall i : \Gamma \vdash e_i :: \mathsf{t}_2}{\Gamma \vdash (\texttt{case}_T\ e\ \texttt{of}(c_1\ x_{1,1}\ \ldots\ x_{n_1,1}\ \texttt{->}\,e_1) \ldots (c_m\ x_{1,m}\ \ldots\ x_{n_m,m}\ \texttt{->}\,e_m)) :: \mathsf{t}_2}$$

Fig. 14. Typing rules for *CH*

C Proofs and Additonal Definitions for the CH-Calculus

C.1 Typing System and Rules for CH

Even though the type system is monomorphic for simplicity, we "overload" the data constructors and thus assume that data types used in case-constructs have a fixed arity, and that the data constructors of every type have a polymorphic type according to the usual conventions. The set of monomorphic types of constructor c is denoted as $\texttt{types}(c)$.

For simplicity, we assume that every variable is explicitly typed: we assume that every variable x has a built-in type, denoted by a global typing function for variables with Γ, i.e. $\Gamma(x)$ is the type of variable x. The notation $\Gamma \vdash e :: \tau$ means that type τ can be derived for expression e using the global typing function Γ. For processes the notation $\Gamma \vdash P :: \texttt{wt}$ means that the process P can be well-typed using the global typing function Γ. The typing rules are given in Fig. 14.

C.2 Proof of Proposition 3.5

Proposition 3.5. *Let P_1, P_2 be well-formed and $P_1 \equiv P_2$. Then $P_1 \sim_c P_2$.*

Proof. Let P_1, P_2 be well-formed, $P_1 \equiv P_2$, and $\mathbb{D} \in PCtxt_{CH}$. We have to show i) $\mathbb{D}[P_1]{\downarrow} \implies \mathbb{D}[P_2]{\downarrow}$; ii) $\mathbb{D}[P_2]{\downarrow} \implies \mathbb{D}[P_1]{\downarrow}$; iii) $\mathbb{D}[P_1]{\Downarrow} \implies \mathbb{D}[P_2]{\Downarrow}$; and iv) $\mathbb{D}[P_2]{\Downarrow} \implies \mathbb{D}[P_1]{\Downarrow}$. The process $\mathbb{D}[P_1]$ is successful iff $\mathbb{D}[P_2]$ is successful, since $\equiv$ cannot remove or introduce a main-thread. For any process P_3, we have:

$\mathbb{D}[P_1] \xrightarrow{sr} P_3$ iff $\mathbb{D}[P_2] \xrightarrow{sr} P_3$, since $\equiv$ is a congruence and since $\xrightarrow{sr}$ is closed w.r.t. $\equiv$. Using both facts, we can use induction on a given reduction sequence starting with $\mathbb{D}[P_i]$ and ending in a successful process, to show that $\mathbb{D}[P_j]$ reduces to a successful process, where $(i, j) \in \{(1, 2), (2, 1)\}$. This shows parts i) and ii).

For the remaining parts, we show $\mathbb{D}[P_1]\uparrow \iff \mathbb{D}[P_2]\uparrow$. We use induction on a given reduction sequence starting with $\mathbb{D}[P_i]$ and ending in must-divergent process to show that $\mathbb{D}[P_j]\uparrow$ where $(i, j) \in \{(1, 2), (2, 1)\}$. The base case holds, since parts i) and ii) imply $\mathbb{D}[P_1]\Uparrow \iff \mathbb{D}[P_2]\Uparrow$ and the induction step again uses the fact, that each reduction step applied to $\mathbb{D}[P_i]$ can also be applied to $\mathbb{D}[P_j]$ for $(i, j) \in \{(1, 2), (2, 1)\}$.

C.3 Embedding *CH* in *CHF*

We briefly recall the program calculus *CHF* [25,26] and then provide an embedding of *CH* into *CHF*. After showing that the embedding is adequate w.r.t. contextual semantics we transfer results on the correctness of program transformations from *CHF* to *CH*. Since the calculus *CHF* is quite similar to the calculus *CH*, we often only comment on the differences between both calculi.

The syntax of processes, expressions, monadic expressions and types is given in Fig. 15. The laws of structural congruence and the syntax of contexts are given in Figs. 16 and 17. The calculus *CHF* uses the same types as the calculus *CH*. Expressions in *CHF* are like expressions in *CH*, with the only difference that the monadic operator `forkIO` is replaced by the monadic operator `future`. While `forkIO` is of type `IO ()` $\to$ `IO ()` the operator `future` is of type `IO t` $\to$ `IO t` for any type `t`. On the process-level *CH* threads $\Leftarrow e$ are replaced by *futures* $x \Leftarrow e$ where x is an introduced variable, called the name of the future. The operational behavior is, that expression e (of type `IO t`) is evaluated concurrently and thereafter the result (of type `t`) is bound to variable x, which makes the value of the concurrent computation accessible. In the standard reduction of the calculus *CHF* (see Fig. 18) this is implemented by the following changes w.r.t. the standard reduction in *CH*: the rule (fork) creates the new future (instead of a new thread) and 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 by the (new) rule (unIO).

The *main thread* in *CHF* is a future and it is written as $x \overset{\text{main}}{\Longleftarrow} e$. Definitions of *functional values*, *monadic values*, and *values* are as in *CH* adapted to the syntax changes. A well-formed process P is *successful*, if $P \equiv \nu x_1. \ldots \nu x_n.(x \overset{\text{main}}{\Longleftarrow} \texttt{return}\ e \mid P')$. The definitions of may-convergence and should-convergence in *CHF* are the same as in *CH* (see Definition 3.2), but adapted to the syntax changes and using the standard reduction defined in Fig. 18. Contextual approximation $\leq_c$ and contextual equivalence $\sim_c$ on *CHF*-processes and expressions are defined analogous to Definition 3.3. Well-formed and structural congruent *CHF*-processes $P_1 \equiv P_2$ are also contextual equivalent (i.e. $P_1 \sim_c P_2$, see [25]).

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

Fig. 15. Syntax of expressions, processes, and types of *CHF*

$$
\begin{aligned}
P_1 \boldsymbol{|} P_2 &\equiv P_2 \boldsymbol{|} P_1 \\
(P_1 \boldsymbol{|} P_2) \boldsymbol{|} P_3 &\equiv P_1 \boldsymbol{|} (P_2 \boldsymbol{|} P_3) \\
(\nu x.P_1) \boldsymbol{|} P_2 &\equiv \nu x.(P_1 \boldsymbol{|} P_2) \text{ if } x \notin FV(P_2) \\
\nu x_1.\nu x_2.P &\equiv \nu x_2.\nu x_1.P \\
P_1 &\equiv P_2 \text{ if } P_1 =_\alpha P_2
\end{aligned}
$$

Fig. 16. Structural congruence of *CHF*

$$
\begin{aligned}
\mathbb{D} \in \mathit{PCtxt}_{\mathit{CHF}} &::= [\cdot] \mid \mathbb{D} \boldsymbol{|} P \mid P \boldsymbol{|} \mathbb{D} \mid \nu x.\mathbb{D} \\
\mathbb{M} \in \mathit{MCtxt}_{\mathit{CHF}} &::= [\cdot] \mid \mathbb{M} \ \texttt{>>=}\ e \\
\mathbb{F} \in \mathit{FCtxt}_{\mathit{CHF}} &::= \mathbb{E} \mid (\texttt{takeMVar}\ \mathbb{E}) \mid (\texttt{putMVar}\ \mathbb{E}\ e) \\
\mathbb{E} \in \mathit{ECtxt}_{\mathit{CHF}} &::= [\cdot] \mid (\mathbb{E}\ e) \mid (\texttt{seq}\ \mathbb{E}\ e) \mid (\texttt{case}\ \mathbb{E}\ \texttt{of}\ \mathit{alts})
\end{aligned}
$$

Fig. 17. *PCtxt*$_{\mathit{CHF}}$-, *MCtxt*$_{\mathit{CHF}}$-, *FCtxt*$_{\mathit{CHF}}$-, *ECtxt*$_{\mathit{CHF}}$-contexts

Monadic Computations

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

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

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

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

(fork) $y \Leftarrow \mathbb{M}[\texttt{future}\ e] \xrightarrow{sr} \nu z.(y \Leftarrow \mathbb{M}[\texttt{return}\ z] \boldsymbol{|} z \Leftarrow e)$ 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) $y \Leftarrow \mathbb{M}[\mathbb{F}[x]] \boldsymbol{|} x = e \xrightarrow{sr} y \Leftarrow \mathbb{M}[\mathbb{F}[e]] \boldsymbol{|} x = e$

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

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

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

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

(dtmvar) $\nu x.\mathbb{D}[y \Leftarrow \mathbb{M}[\texttt{takeMVar x}] \mid x \,\mathbf{m}\, e] \rightarrow \nu x.\mathbb{D}[y \Leftarrow \mathbb{M}[\texttt{return}\ e] \mid x \,\mathbf{m}\, -]$,
if for $\forall \mathbb{D}' \in PCtxt_{CHF}$ the first execution of $(\texttt{takeMVar}\ x)$ is in thread y for all sr-reductions starting with $\mathbb{D}'[\nu x.(\mathbb{D}[y \Leftarrow \mathbb{M}[\texttt{takeMVar}\ x] \mid x \,\mathbf{m}\, e])]$

(dpmvar) $\nu x.\mathbb{D}[y \Leftarrow \mathbb{M}[\texttt{putMVar}\ x\ e] \mid x \,\mathbf{m}\, -] \rightarrow \nu x.\mathbb{D}[y \Leftarrow \mathbb{M}[\texttt{return ()}] \mid x \,\mathbf{m}\, e]$,
if $\forall \mathbb{D}' \in PCtxt_{CHF}$ and any e' the first execution of $(\texttt{putMVar}\ x\ e')$for all sr-reductions starting with $\mathbb{D}'[\nu x.(\mathbb{D}[y \Leftarrow \mathbb{M}[\texttt{putMVar}\ x\ e] \mid x \,\mathbf{m}\, -])]$ is in thread y

Fig. 19. Transformations (gc), (dtmvar), and (dpmvar)

Program transformations η in *CHF* are binary relations between *CHF*-processes, where η is *correct* iff $\eta \subseteq \sim_c$. We define garbage collection (gc) and deterministic variants of (tmvar) and (pmvar) for the calculus *CHF* in Fig. 19.

Proposition C.1 ([25]). *The transformations (lunit), (nmvar), (fork), (unIO), (cpce), (mkbinds), (beta), (case), (seq) (gc), (dtmvar), and (dpmvar) are correct program transformations, whereas (pmvar) and (tmvar) in general are not correct as program transformations.*

We define an embedding ι which embeds *CH* into *CHF*:

Definition C.2. *We define the translation $\iota : CH \rightarrow CHF$. For most of the cases ι is applied homomorphically over the term structure, without changing the syntax, i.e. for variables r and nullary constructors r: $\iota(r) := r$ and for any syntactic operator f and arguments $r_1, \cdots, r_n$: $\iota(f\ r_1\ \ldots\ r_n) := f\ \iota(r_1)\ \ldots\ \iota(r_n)$. We now explicitly list the exceptional cases: For expressions* `forkIO` *e we define:*

$$\iota(\texttt{forkIO}\ e) := (\texttt{future}\ \iota(e)) \gg \texttt{return}\ \mathit{()}$$

The exceptional cases on the process level are

$$\iota(\Leftarrow e) := \nu x.x \Leftarrow \iota(e) \ \mathit{and}\ \iota(\stackrel{\mathsf{main}}{\Longleftarrow} e) := \nu x.x \stackrel{\mathsf{main}}{\Longleftarrow} \iota(e)$$

where for both cases, we assume that x is chosen (fresh) such that x does not occur in $\iota(e)$. On types, the translation ι is the identity.

Proposition C.3. *Assume that the countable infinite set of variables in CH is Var_{CH} and in CHF is $Var_{CHF} = Var_{CH} \cup Var'$ where $Var_{CH} \cap Var' = \emptyset$, Var' is countably infinite and translation ι always uses a variable of Var' for translating $\Leftarrow e$ or $\stackrel{\mathsf{main}}{\Longleftarrow} e$. Then the translation ι is compositional, i.e. $\iota(C[e]) = \iota(C)[\iota(e)]$.*

For analyzing the correspondence between the sr-reductions in *CH* and *CHF* w.r.t. ι, an observation is that for all $\mathbb{D} \in PCtxt_{CH}$: $\iota(\mathbb{D}) \in PCtxt_{CHF}$; for all $\mathbb{M} \in MCtxt_{CH}$: $\iota(\mathbb{M}) \in MCtxt_{CHF}$; for all $\mathbb{E} \in ECtxt_{CH}$: $\iota(\mathbb{E}) \in ECtxt_{CHF}$; and for all $\mathbb{F} \in FCtxt_{CH}$: $\iota(\mathbb{F}) \in FCtxt_{CHF}$. We analyze the sr-reduction w.r.t. ι.

Lemma C.4. *Let P, P' be CH-processes with $P \xrightarrow{sr} P'$. Then either $\iota(P) \xrightarrow{sr} \iota(P')$ or $\iota(P) \xrightarrow{sr,3} \iota(P')$ where in the latter case $\iota(P) \sim_c \iota(P')$.*

Proof. We consider the different reduction rules. For rules (lunit), (tmvar), (pmvar), (nmvar), and for all functional evaluations, we have: if $P \xrightarrow{sr} P$ then $\iota(P) \xrightarrow{sr} \iota(P')$. For `(fork)`, let $P = \mathbb{D}[\Leftarrow \mathbb{M}[\texttt{forkIO}\ e]] \xrightarrow{sr} \mathbb{D}[\Leftarrow \mathbb{M}[\texttt{return}\ ()] \mid\ \Leftarrow e] = P'$. Then we can reduce $\iota(P)$ as follows:

$$\begin{aligned}
&\iota(P) = \iota(\mathbb{D})[\nu x.x \Leftarrow \texttt{future}\ \iota(e) \gg\!= \lambda\_.\texttt{return ()}]\\
&\xrightarrow{sr,fork} \iota(\mathbb{D})[\nu x.\nu z.x \Leftarrow \texttt{return}\ z \gg\!= \lambda\_.\texttt{return ()} \mid z \Leftarrow \iota(e)]\\
&\xrightarrow{sr,lunit} \iota(\mathbb{D})[\nu x.\nu z.x \Leftarrow (\lambda\_.\texttt{return ()})\ z \mid z \Leftarrow \iota(e)]\\
&\xrightarrow{sr,beta} \iota(\mathbb{D})[\nu x.\nu z.x \Leftarrow \texttt{return ())} \mid z \Leftarrow \iota(e)] \equiv \iota(P')
\end{aligned}$$

Theorem C.1 implies $\iota(P) = \iota(P')$.

Lemma C.5. *$P \in Proc_{CH}$ is successful iff $\iota(P)$ is successful.*

Proof. If $P \in Proc_{CH}$ is successful, then $P \equiv \mathbb{D}[\overset{\mathsf{main}}{\Longleftarrow} \texttt{return ()}]$ and $\iota(P) = \iota(\mathbb{D})[\nu x.x \overset{\mathsf{main}}{\Longleftarrow} \texttt{return ()}]$. If $\iota(P)$ is successful, then $\iota(P) = \mathbb{D}[x \overset{\mathsf{main}}{\Longleftarrow} \texttt{return ()}]$ and thus $P \equiv \mathbb{D}'[\Leftarrow \texttt{return ()}]$ for some $\mathbb{D}' \in PCtxt_{CH}$.

Proposition C.6. *Let $P \in Proc_{CH}$ such that $P\downarrow$. Then $\iota(P)\downarrow$.*

Proof. By induction on $P \xrightarrow{sr,*} P'$ where P' is successful. The base case holds by Lemma C.5. For the induction step, let $P \xrightarrow{sr} P'' \xrightarrow{sr,*} P'$. The induction hypothesis shows $\iota(P'')\downarrow$ and Lemma C.4 shows $\iota(P) \xrightarrow{sr,*} \iota(P'')$. Thus $\iota(P)\downarrow$.

Checking all cases of the standard reduction in *CHF* together with inspecting the images of the translation ι shows:

Lemma C.7. *Let $P \in Proc_{CH}$, $\iota(P) \xrightarrow{sr,a} P'$, and $a \notin \{fork, unIO\}$. Then there exists P'' with $P \xrightarrow{sr} P''$ such that $\iota(P'') = P'$, i.e. the diagram shown on the right side holds.*

$$\begin{tikzcd}
P \arrow[r, "\iota"] \arrow[d, dashed, "sr{,}a"'] & \iota(P) \arrow[d, "sr{,}a"] \\
P'' \arrow[r, dashed, "\iota"'] & P'
\end{tikzcd}$$

Lemma C.8. *Let $P \in Proc_{CH}$ such that $\iota(P) \xrightarrow{sr,fork} P'$. Then there exists a CH-process P'' where $P \xrightarrow{sr,fork} P''$ and $P' \xrightarrow{sr,*} \iota(P'')$. The equivalence $\iota(P'') \sim_c P'$ holds. The diagram on the right hand side shows the given and existentially quantified reductions and translations steps.*

$$\begin{tikzcd}
P \arrow[r, "\iota"] \arrow[d, dashed, "sr{,}a"'] & \iota(P) \arrow[d, "sr{,}a"] \\
P'' \arrow[dr, dashed, "\iota"'] & P' \arrow[d, dashed, "sr{,}*"] \\
 & \iota(P'')
\end{tikzcd}$$

Proof. The (sr,fork)-step in *CHF* evaluates a `future`-operation that must stem from translating a `forkIO`. Since $\iota(\texttt{forkIO}\ e) = \texttt{future}\ \iota(e) \gg \texttt{return ()}$, the reduction $\iota(P) \xrightarrow{sr,fork} P'$ can be extended by two reduction steps $P' \xrightarrow{sr,lunit} \cdot \xrightarrow{sr,beta} P'''$ such that $P \xrightarrow{sr,fork} P''$ and $\iota(P'') = P'''$. Since (sr,beta) and (sr,fork) are correct in *CHF* (Proposition C.1), we have $P''' \sim_c P''$. □

The two reductions leading from P' to $\iota(P'')$ in the previous lemma, are correct program transformations, and they are deterministic for the corresponding thread (i.e. for this thread, no other reduction is possible, even if other components or threads change (for instance, the content of MVars). Thus, we can safely add all the missing reduction steps corresponding to Lemma C.8 to a given sequence $\iota(P) \xrightarrow{sr,*} P_0$ resulting in a sequence $\iota(P) \xrightarrow{sr,*} P_1$ such that i) $P_0 \sim_c P_1$ and ii) P_1 is successful if P_0 is successful.

Lemmas C.7 and C.8 do not cover the case where $\iota(P) \xrightarrow{sr,unIO} P'$ for some $P \in \mathit{Proc}_{CH}$. In this case $P = \mathbb{D}[\Leftarrow \texttt{return ()}]$ must hold, and $\iota(P) = \iota(\mathbb{D})[\nu x.x \Leftarrow \texttt{return ()}] \xrightarrow{sr,unIO} \iota(\mathbb{D})[\nu x.x = \texttt{()}]$. Since future x cannot be used in the context $\iota(\mathbb{D})$, it is irrelevant, if the (sr,unIO)-reduction is performed or is not performed. I.e., given a sequence $\iota(P) \xrightarrow{sr,*} P_1$, we can modify the sequence, by removing all (sr,unIO)-reduction, resulting in a sequence $\iota(P) \xrightarrow{sr,*} P_2$ such that i) $P_1 \sim_c P_2$ and ii) P_1 is successful if and only if P_2 is successful.

Definition C.9. *Let $Red = \iota(P) \xrightarrow{sr,*} P'$ be a CHF-standard reduction sequence for $P \in \mathit{Proc}_{CH}$. We say that Red is ι-*normalized *if the two described transformations on reduction sequences (adding reduction steps corresponding to Lemma C.8 and removing (sr,unIO)-reductions) are not applicable to Red.*

As argued above, the following lemma holds:

Lemma C.10. *For every sequence $\iota(P) \xrightarrow{sr,*} P'$ with $P \in \mathit{Proc}_{CH}$, there exists a ι-normalized reduction sequence $\iota(P) \xrightarrow{sr,*} P''$ such that i) $P' \sim_c P''$ and ii) P'' is successful if P' is successful.*

Proposition C.11. *Let $P \in \mathit{Proc}_{CH}$ and $\iota(P)\downarrow$. Then $P\downarrow$.*

Proof. Let $\iota(P) \xrightarrow{sr,*} P_0$ where P_0 is successful. By Lemma C.10 there exists a ι-normalized reduction sequence $\iota(P) \xrightarrow{sr,n} P_1$ where P_1 is successful and $n \geq 0$. By induction on n we show $P'\downarrow$. Lemma C.5 covers the case $n = 0$. If $n > 0$ then $\iota(P) \xrightarrow{sr,n} P_1$. Since the given sequence is ι-normalized, we can either apply the commutation diagram from Lemma C.7 or the diagram from Lemma C.8 to a prefix of $\iota(P) \xrightarrow{sr,n} P_1$ such that $P \xrightarrow{sr} P''$ and $\iota(P'') \xrightarrow{sr,n'} P_0$ where $n' < n$ and $\iota(P'') \xrightarrow{sr,n'} P_1$ is ι-normalized and thus we can apply the induction hypothesis which shows $P''\downarrow$ and thus $P\downarrow$. □

By Propositions C.6 and C.11 we also have:

Corollary C.12. *Let $P \in \mathit{Proc}_{CH}$. Then $\iota(P)\Uparrow \iff P\Uparrow$*

Proposition C.13. *Let $P \in \mathit{Proc}_{CH}$ such that $P\uparrow$. Then $\iota(P)\uparrow$.*

Proof. By induction on length n of a reduction $P \xrightarrow{sr,n} P'$ where $P'\Uparrow$. Corollary C.12 covers the base case. For the induction step, let $P \xrightarrow{sr} P'' \xrightarrow{sr,*} P'$. The induction hypothesis shows $\iota(P'')\uparrow$, Lemma C.4 shows that $\iota(P) \xrightarrow{sr,*} \iota(P'')$, and thus $\iota(P)\uparrow$. □

Proposition C.14. *Let $P \in \mathit{Proc}_{CH}$ and $\iota(P)\uparrow$. Then $P\uparrow$.*

Proof. Let $\iota(P) \xrightarrow{sr,*} P_0$ where $P_0\Uparrow$. By Lemma C.10 there exists a ι-normalized reduction sequence $\iota(P) \xrightarrow{sr,n} P_1$ where $P_1\Uparrow$ and $n \geq 0$. We use induction on n to show $P\uparrow$. If $n = 0$, then Corollary C.12 shows that $P\Uparrow$. If $n > 0$, then $\iota(P) \xrightarrow{sr,n} P_1$. Since the sequence is ι-normalized, we can either apply the commutation diagram from Lemma C.7 or the diagram from Lemma C.8 to a prefix of $\iota(P) \xrightarrow{sr,n} P_1$ such that $P \xrightarrow{sr} P''$ and $\iota(P'') \xrightarrow{sr,n'} P_0$ where $n' < n$ and also $\iota(P'') \xrightarrow{sr,n'} P_1$ is ι-normalized and thus we can apply the induction hypothesis which shows $P''\uparrow$ and thus also $P\uparrow$. □

Propositions C.6, C.11, C.13 and C.14 show:

Theorem C.15. *The translation ι is convergence equivalent.*

Theorem C.16. *The translation ι is adequate, i.e. for all processes $P_1, P_2 \in \mathit{Proc}_{CH}$: $\iota(P_1) \sim_c \iota(P_2) \implies P_1 \sim_c P_2$ and for all expressions $e_1, e_2 \in \mathit{Expr}_{CH}$: $\iota(e_1) \sim_c \iota(e_2) \implies e_1 \sim_c e_2$.*

Proof. This holds, since ι is compositional and convergence equivalent. We only show the part for processes. Let $\xi \in \{\downarrow, \Downarrow\}, (i,j) \in \{(1,2),(2,1)\}$. Let $\iota(P_i) \sim_c \iota(P_j)$ and $\mathbb{D} \in \mathit{PCtxt}_{CH}$ such that $\mathbb{D}[P_i]\xi$. Then $\iota(\mathbb{D}[P_i]) = \iota(\mathbb{D})[\iota(P_i)]$ (since ι is compositional) and thus $\iota(\mathbb{D})[\iota(P_i)]\xi$ (since ι is convergence equivalent)s. Since $\iota(P_i) \sim_c \iota(P_j)$, this implies $\iota(\mathbb{D})[\iota(P_j)]\xi$ which implies $\iota(\mathbb{D}[P_j])\xi$ (since ι is compositional) and thus $\mathbb{D}[P_j]\xi$ (since ι is convergence equivalent). □

Proposition 3.6. *The transformations (lunit), (nmvar), (fork), (cpce), (mkbinds), (beta), (case), (seq), (gc), (dtmvar), and (dpmvar) are correct in CH.*

Proof. For all mentioned transformations η, we have: if $P \xrightarrow{\eta} P'$ then $\iota(P) \sim_c \iota(P')$ by Proposition C.1, which implies $P \sim_c P$ by Theorem C.16. □

D Barbed Convergence Testing and Equivalence

In [27], the following claim is proved, where σ are name-to-name substitutions:

Theorem D.1 ([27]). *For all processes $P, Q \in \Pi_{\mathtt{Stop}}$:*

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

For stop-free processes, contextual equivalence coincides with so-called barbed testing equivalence, where the observation is whether a process may- or should-reduces to a process that has a free input on a fixed channel name (see [27]):

Definition D.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 $P{\Rsh}^{x}$) iff $P \equiv \nu\mathcal{X}.(x(y)P' \mid P'')$ where $x \notin \mathcal{X}$. We write $P\downharpoonright_x$ iff there exists P' such that $P \xrightarrow{sr,*} P'$ and $P'{\Rsh}^{x}$. We write $P \downharpoonright\!\downharpoonright_x$ iff for all P' with $P \xrightarrow{sr,*} P'$ also $P'\downharpoonright_x$ holds. We write $P \upharpoonright\!\upharpoonright_x$ iff $P\downharpoonright_x$ does not hold, and $P\upharpoonright_x$ iff $P \downharpoonright\!\downharpoonright_x$ 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 $\leq_{c,barb} := \leq_{c,\downharpoonright_x} \cap \leq_{c,\downharpoonright\!\downharpoonright_x}$ and $\sim_{c,barb} := \leq_{c,barb} \cap (\leq_{c,barb})^{-1}$ where for $\xi \in \{\downharpoonright_x, \downharpoonright\!\downharpoonright_x, \upharpoonright_x, \upharpoonright\!\upharpoonright_x\}$ and $P, Q \in \Pi$, $P \leq_{c,\xi} Q$ holds iff for all contexts $C \in \Pi : C[P]\xi \implies C[Q]\xi$.*

Theorem D.4 ([27]). *For all processes $P, Q \in \Pi$: $P \leq_{c,barb} Q \iff P \leq_c Q$, and hence also $P \sim_{c,barb} Q \iff P \sim_c Q$.*