Correctly Implementing Synchronous Message Passing in the Pi-Calculus By Concurrent Haskell's MVars

Manfred Schmidt-Schauß

Goethe-University, Frankfurt am Main, Germany

schauss@ki.cs.uni-frankfurt.de

David Sabel

LMU, Munich, Germany

david.sabel@ifi.lmu.de [*]

Comparison of concurrent programming languages and correctness of program transformations in concurrency are the focus of this research. As criterion we use contextual semantics adapted to concurrency, where may- as well as should-convergence are observed. We investigate the relation between the synchronous pi-calculus and a core language of Concurrent Haskell (CH). The contextual semantics is on the one hand forgiving with respect to the details of the operational semantics, and on the other hand implies strong requirements for the interplay between the processes after translation. Our result is that CH embraces the synchronous pi-calculus. Our main task is to find and prove correctness of encodings of pi-calculus channels by CH's concurrency primitives, which are MVars. They behave like (blocking) 1-place buffers modelling the shared-memory. The first developed translation uses an extra private MVar for every communication. We also automatically generate and check potentially correct translations that reuse the MVars where one MVar contains the message and two additional MVars for synchronization are used to model the synchronized communication of a single channel in the pi-calculus. Our automated experimental results lead to the conjecture that one additional MVar is insufficient.

1 Introduction

Our goals are the comparison of programming languages, correctness of transformations, compilation and optimization of programs, in particular of concurrent programs. We already used the contextual semantics of concurrent (functional) programming languages to effectively verify correctness of transformations [16, 23, 24], also under the premise not to worsen the runtime [30]. We propose to test may- and should-convergence in the contextual semantics, since, in particular, it rules out transformations that transform an always successful process into a process that may run into an error, for example a deadlock. There are also other notions of program equivalence in the literature, like bisimulation based program equivalences [27], however, these tend to take also implementation specific behavior of the operational semantics into account, whereas contextual equivalence abstracts from the details of the executions.

In [28, 31] we developed notions of correctness of translations w.r.t. contextual semantics, and in [32] we applied them in the context of concurrency, but for quite similar source and target languages. In this paper we translate a synchronous message passing model into a shared memory model, namely a synchronous π-calculus into a core-language of Concurrent Haskell, called *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 $C[P]$ – i.e. P plugged into a program context – 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*[1] ($P{\Downarrow}$) – every intermediate state of a reduction sequence may-converges. For two processes P and Q, $P \leq_c Q$ holds iff for all contexts $C[\cdot]$: $(C[P]{\downarrow} \implies C[Q]{\downarrow})$, and P and Q are contextually equivalent, $P \sim_c Q$, iff $P \leq_c Q$ and $Q \leq_c P$. Showing equal expressivity of two (concurrent) calculi by a translation τ requires that may- and should-convergence make sense in each calculus. Important properties are convergence-equivalence (may- and should-convergencies are preserved and reflected by the translation) and adequacy (see Definition 4.4), which holds if $\tau(P) \leq_{c,CH} \tau(Q)$ implies $P \leq_{c,\pi} Q$, for all π-calculus processes P, Q. Full-abstraction, i.e. $\forall P, Q : \tau(P) \leq_c \tau(Q)$ iff $P \leq_c Q$, only holds if the two calculi are more or less the same.

Source and Target Calculi. The well-known π-calculus [15, 14, 27] is a minimal model for *mobile and concurrent processes*. Dataflow is expressed by passing messages between them via named channels, where messages are channel names. Processes and links between processes can be dynamically created and removed which 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 [21], and distributed computing [13, 7]. The π-calculus also permits the study of intrinsic principles and semantics of concurrency and the inherent nondeterministic behavior of mobile and communicating processes. We investigate a variant of the π-calculus which is the synchronous π-calculus with replication, but without sums, matching operators, or recursion. To observe termination of a process, the calculus has a constant `Stop` which signals successful termination.

The calculus *CH*, a core language of Concurrent Haskell, 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 modelled by MVars (mutable variables) which are one-place buffers that can be either filled or empty. The operation `takeMVar` tries to empty a filled MVar and blocks if the MVar is already empty. The operation `putMVar` tries to fill an empty MVar and blocks if the MVar is already filled. The calculus *CH* is a variant (or a subcalculus) of the calculus *CHF* [23, 24] which extends Concurrent Haskell with futures. A technical 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 encode the synchronous communication of the π-calculus. The problem is that the MVars in *CH* have an asynchronous behavior (communication has to be implemented in two steps: the sender puts the message into an MVar, which is later taken by the receiver). To implement synchronous communication, the weaker synchronisation 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 [20], 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 and per communication to ensure that no other process can interfere with the interaction. A similar idea was used in [12, 3] for keeping channel names private in a different scenario (see [10, 9] for recent treatments of these encodings). We prove that the translation τ_0 is correct. Since we are also interested in simpler translations, we looked for correct translations with a fixed and static number of MVars per channel in the π-calculus. Since this

---

[1]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. [32]).

task is too complex and error-prone for hand-crafting, we automated it by implementing a tool to rule out incorrect translations[2]. Thereby we fix the MVars used for every channel: a single MVar for exchanging the channel-name and perhaps several additional MVars of unit type to perform checks whether the message was sent or received (we call them check-MVars, they behave like binary semaphores that are additionally blocking for signal-operations on an unlocked semaphore). The outcomes of our automated search are: a further correct translation that uses two check-MVars, where one is used as a mutex between all senders or receivers on one channel, and further correct translations using three additional MVars where the filling and emptying operations for each MVar need 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 result is convergence-equivalence and adequacy of the open translation τ (Theorems 4.5 and 4.8), translating the π-calculus into CH. The comparison of the π-calculus with a concurrent programming language (here CH) using contextual semantics for may- and should-convergence in both calculi exhibits that the π-calculus is embeddable in CH where we can prove that the semantical properties of interest are kept. The adaptation of the adequacy and full abstraction notions (Definition 4.4) for open processes is a helpful technical extension of our work in [28, 31].

We further define a general formalism for the representation of translations with global names and analyze different classes of such translations using an automated tool. In particular, we show correctness of two particular translations in Theorems 5.9 and 5.12. 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, since there are apparently correct (and simple) translations which were wrong. Our automated tool helped us to rule out wrong translations and to find potentially correct ones.

Discussion of Related Work on Characterizing Encodings. There are five criteria for valid translations resp. encodings proposed and discussed in [11, 9], which mainly restrict the translations w.r.t. language syntax and reduction semantics of the source and target language, called: compositionality, name invariance, operational correspondence, divergence reflection and success sensitiveness. Compositionality and name invariance restrict the syntactic form of the translated processes; operational correspondence means that the transitive closure of the reduction relation is transported by the translation, modulo the syntactic equivalence; and divergence reflection and success sensitiveness are conditions on the semantics.

In our approach, we define semantical congruence (and precongruence) relations on the source and target language. Thus the first two conditions are not part of our notion of contextual equivalence, however, may be used as restrictions in showing non-encodability. We also omit the third condition and only use stronger variants of the fourth and fifth condition. Convergence equivalence as a tool for finding out may-and should-convergence is our replacement of Gorla's divergence reflection and success sensitiveness. We do not define an infinite reduction sequence as an error, which has as nice consequence that synchronization could be implemented by busy-wait techniques.

Further Related Work. Encodings of synchronous communication by asynchronous communication using a private name mechanism are given in [12, 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 our setting is different, since our target language is Concurrent Haskell. Encodings between π-calculi with synchronous and with asynchronous communication were, for instance, already considered in [12, 3, 19, 18]

---

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

$P, Q \in \Pi_{\mathtt{Stop}} ::= \nu x.P \mid \overline{x}y.P \mid x(y).P \mid !P \mid P \mathbin{\mathbf{|}} Q \mid 0 \mid \mathtt{Stop}$
$C \in \Pi_{\mathtt{Stop},C} ::= [\cdot] \mid \overline{x}(y).C \mid x(y).C \mid C \mathbin{\mathbf{|}} P \mid P \mathbin{\mathbf{|}} C \mid !C \mid \nu x.C$
$\mathbb{D} \in PCtxt_\pi ::= [\cdot] \mid \mathbb{D} \mathbin{\mathbf{|}} P \mid P \mathbin{\mathbf{|}} \mathbb{D} \mid \nu x.\mathbb{D}.$

Figure 1: Syntax of processes $\Pi_{\mathtt{Stop}}$, process contexts $\Pi_{\mathtt{Stop},C}$ and reduction contexts $PCtxt_\pi$ where x, y are names.

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

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

Figure 2: Reduction rule and standard reduction in $\Pi_{\mathtt{Stop}}$

$$P \equiv Q, \text{ if } P =_\alpha Q$$
$$P \mathbin{\mathbf{|}} (Q \mathbin{\mathbf{|}} R) \equiv (P \mathbin{\mathbf{|}} Q) \mathbin{\mathbf{|}} R$$
$$\nu x.(P \mathbin{\mathbf{|}} Q) \equiv P \mathbin{\mathbf{|}} \nu x.Q, \text{if } x \notin FN(P)$$
$$P \mathbin{\mathbf{|}} 0 \equiv P$$
$$\nu x.0 \equiv 0$$
$$\nu x.\mathtt{Stop} \equiv \mathtt{Stop}$$
$$\nu x.\nu y.P \equiv \nu y.\nu x.P$$
$$P \mathbin{\mathbf{|}} Q \equiv Q \mathbin{\mathbf{|}} P$$
$$!P \equiv P \mathbin{\mathbf{|}} !P$$

Figure 3: Structural congruence in $\Pi_{\mathtt{Stop}}$

where encodability results are obtained for the π-calculus without sums [12, 3], while in [18, 19] the expressive power of synchronous and asynchronous communication in the π-calculus with *mixed sums* was compared and non-encodability is a main result. Translations of the π-calculus into programming calculi and logical systems are given in [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 [33] 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., [17] 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 the two calculi. In [4] an embedding of a session π-calculus into ReactiveML is given and operational correspondence between the two languages is shown. Encodings of CML-events in Concurrent Haskell using MVars are published in [22, 5]. This approach is more high-level than ours (since it considers events, while we focus the plain π-calculus). In [5] correctness of a distributed protocol for selective-communication w.r.t. an excerpt of CML is shown and a correct implementation of the protocol in the π-calculus is given. The protocol is implemented in Concurrent-Haskell, but no correctness of this part is shown, since [5] 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 the source and target calculi in Sections 2 and 3, the translation using private names in Section 4, and in Section 5 we treat translations with global names. We conclude and discuss future work in Section 6. Due to space constraints most proofs are in the technical report [29].

2 The π-Calculus with Stop

We explain the synchronous π-calculus [15, 14, 27] without sums, with replication, extended with a constant $\mathtt{Stop}$ [25], that signals successful termination. The π-calculus with $\mathtt{Stop}$ and the π-calculus without $\mathtt{Stop}$ but with so-called barbed convergences [26] are equivalent w.r.t. contextual semantics [29]. Thus, adding the constant $\mathtt{Stop}$ is not essential, but our arguments are easier to explain with $\mathtt{Stop}$.

Definition 2.1 (Calculus $\Pi_{\mathtt{Stop}}$)**.** *Let $\mathcal{N}$ be a countable set of (channel) names. The syntax of* processes *is shown in Fig. 1. Name restriction $\nu x.P$ restricts the scope of name x to process P, $P \mathbin{\mathbf{|}} Q$ 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 then becoming P, the process $x(y).P$ waits on channel x to receive input, after receiving the input z, the process turns*

into $P[z/y]$ (where $P[z/y]$ is the substitution of all free occurrences of name y by name z in process P), the process !P is 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, and Stop *is a process constant that signals success. We sometimes write $x(y)$ instead of $x(y).0$ as well as $\overline{x}y$ instead of $\overline{x}y.0$.*

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 $\Pi_{\texttt{Stop}}^c$ be the closed processes in $\Pi_{\texttt{Stop}}$. Structural congruence $\equiv$ is the least congruence satisfying the laws shown in Fig. 3.* Process contexts $\Pi_{\texttt{Stop},C}$ *and* reduction contexts $PCtxt_\pi$ *are defined in Fig. 1. Let $C[P]$ be the substitution of the hole $[\cdot]$ in C by P. The reduction rule $\xrightarrow{ia}$ performs* interaction *and standard reduction $\xrightarrow{sr}$ is its closure w.r.t. reduction contexts and $\equiv$ (see Fig. 2). Let $\xrightarrow{sr,n}$ denote n $\xrightarrow{sr}$-reductions and $\xrightarrow{sr,*}$ denotes the reflexive-transitive closure of $\xrightarrow{sr}$. A process $P \in \Pi_{\texttt{Stop}}$ is* successful, *if $P \equiv \mathbb{D}[\texttt{Stop}]$ for some $\mathbb{D} \in PCtxt_\pi$.*

Remark 2.2. *We do not include "new" laws for structural congruences on the constant* Stop, *like* Stop | Stop *equals* Stop, *since this would require to re-develop a lot of theory known from the π-calculus without* Stop. *In our view,* Stop *is a mechanism for a notion of success 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 those equations* on the semantic level. *(i.e. w.r.t. $\sim_c$ as defined below in Definition 2.5).*

As an example for a reduction sequence, consider sending name y over channel x and then sending name u over channel y: $(x(z).\overline{z}u.0 \mid \overline{x}y.y(x).0) \xrightarrow{ia} (\overline{z}u.0[y/z] \mid y(x).0) \equiv (y(x).0 \mid \overline{y}u.0) \xrightarrow{ia} (0 \mid 0) \equiv 0$.

For the semantics of processes, we observe whether standard reductions successfully terminate or not. Since reduction is nondeterministic, we test whether there exists a successful reduction sequence (may-convergence), and we test whether all reduction possibilities are successful (should-convergence).

Definition 2.3. *Let P be a $\Pi_{\texttt{Stop}}$-process. We say process P is* may-convergent *and write $P{\downarrow}$, if and only if there is a successful process P' with $P \xrightarrow{sr,*} P'$. We say P is* should-convergent *and write $P{\Downarrow}$ if and only if for all P': $P \xrightarrow{sr,*} P'$ implies $P'{\downarrow}$. If P is not may-convergent, then P we say is* must-divergent *(written $P{\Uparrow}$). If P is not should-convergent, then we say it is* may-divergent *(written $P{\uparrow}$).*

Example 2.4. *The process $P := \nu x,y.(x(z).0 \mid \overline{x}y.\texttt{Stop})$ is may-convergent ($P{\downarrow}$) and should-convergent ($P{\Downarrow}$), since $P \xrightarrow{sr} 0 \mid \texttt{Stop}$ is the only $\xrightarrow{sr}$-sequence for P. The process $P' := \nu x,y.(x(z).0 \mid \overline{x}y.0)$ is may- and must-divergent (i.e. $P'{\uparrow}$ and $P'{\Uparrow}$), since $P' \xrightarrow{sr} 0$ is the only $\xrightarrow{sr}$-sequence for P'.*
For $P'' := \nu x,y.(\overline{x}y.0 \mid x(z).\texttt{Stop} \mid x(z).0)$, we have $P'' \xrightarrow{sr} \nu x,y.(\texttt{Stop} \mid x(z).0)$ and $P'' \xrightarrow{sr} \nu x,y.x(z).\texttt{Stop}$. The first result is successful, and the second result is not successful. Hence, for P'' we have $P''{\downarrow}$ and $P''{\uparrow}$.

Should-convergence implies may-convergence, and must-divergence implies may-divergence.

Definition 2.5. *For $P,Q \in \Pi_{\texttt{Stop}}$ and observation $\xi \in \{{\downarrow},{\Downarrow},{\uparrow},{\Uparrow}\}$, we define $P \leq_\xi Q$ iff $P\xi \implies Q\xi$. The ξ-contextual preorders $\leq_{c,\xi}$ and then ξ-contextual equivalences $\sim_{c,\xi}$ are defined as*

$$P \leq_{c,\xi} Q \text{ iff } \forall C \in \Pi_{\texttt{Stop},C} : C[P] \leq_\xi C[Q] \quad and \quad P \sim_{c,\xi} Q \text{ iff } P \leq_{c,\xi} Q \wedge Q \leq_{c,\xi} P$$

Contextual equivalence *of $\Pi_{\texttt{Stop}}$-processes is defined as $P \sim_c Q$ iff $P \sim_{c,{\downarrow}} Q \wedge P \sim_{c,{\Downarrow}} Q$.*

Example 2.6. *For $Q := \nu x,y.(\overline{x}y.0 \mid x(z).\texttt{Stop} \mid x(z).0)$, we have* Stop $\sim_{c,{\downarrow}} Q$ *(which can be proved using the methods in [25]), but* Stop $\not\sim_c Q$, *since* Stop${\Downarrow}$ *and $Q{\uparrow}$ and thus* Stop $\not\sim_{c,{\Downarrow}} Q$. *Note that $\leq_{c,{\Downarrow}} = \leq_c$ holds in $\Pi_{\texttt{Stop}}$, since there is a context C such that for all processes P: $C[P]{\Downarrow} \iff P{\downarrow}$ (see [25]). For instance, the equivalence $0 \sim_{c,{\Downarrow}} Q$ does not hold, since $!0{\Uparrow}$ and $!Q{\Downarrow}$ and thus the context $C =![\cdot]$ distinguishes 0 and Q w.r.t. should-convergence.*

Contextual preorder and equivalence are (pre)-congruences. Contextual preorder remains unchanged if observation is restricted to closing contexts:

Lemma 2.7. *Let $\xi \in \{{\downarrow},{\uparrow},{\Downarrow},{\Uparrow}\}$, P,Q be $\Pi_{\texttt{Stop}}$-processes. Then $P \leq_{c,\xi} Q$ if, and only if $\forall C \in \Pi_{\texttt{Stop},C}$ such that $C[P]$ and $C[Q]$ are closed: $C[P] \leq_\xi C[Q]$.*

$$P \in Proc_{CH} ::= (P_1 \mathbin{|} P_2) \mid \Leftarrow e \mid \nu x.P \mid x\,\mathbf{m}\,e \mid x\,\mathbf{m}\,{-} \mid x = e$$
$$e \in Expr_{CH} ::= x \mid \lambda x.e \mid (e_1\ e_2) \mid c\ e_1 \ldots e_{\mathrm{ar}(c)} \mid \mathtt{letrec}\ x_1{=}e_1,\ldots,x_n{=}e_n\ \mathtt{in}\ e \mid m \mid \mathtt{seq}\ e_1\ e_2$$
$$\mid \mathtt{case}_T\ e\ \mathtt{of}\ (c_{T,1}\,x_1 \ldots x_{\mathrm{ar}(c_{T,1})} \text{->} e_1) \ldots (c_{T,|T|}\,x_1 \ldots x_{\mathrm{ar}(c_{T,|T|})} \text{->} e_{|T|})$$
$$m \in MExpr_{CH} ::= \mathtt{return}\,e \mid e \gg\!\!= e' \mid \mathtt{forkIO}\,e \mid \mathtt{takeMVar}\,e \mid \mathtt{newMVar}\,e \mid \mathtt{putMVar}\,e\,e'$$
$$\mathfrak{t} \in Typ_{CH} ::= \mathtt{IO}\ \mathfrak{t} \mid (T\ \mathfrak{t}_1 \ldots \mathfrak{t}_n) \mid \mathtt{MVar}\ \mathfrak{t} \mid \mathfrak{t}_1 \to \mathfrak{t}_2$$
$$\mathbb{D} \in PCtxt_{CH} ::= [\cdot] \mid \mathbb{D} \mathbin{|} P \mid P \mathbin{|} \mathbb{D} \mid \nu x.\mathbb{D} \qquad \mathbb{E} \in ECtxt_{CH} ::= [\cdot] \mid (\mathbb{E}\,e) \mid (\mathtt{seq}\,\mathbb{E}\,e) \mid (\mathtt{case}\,\mathbb{E}\,\mathtt{of}\,alts)$$
$$\mathbb{M} \in MCtxt_{CH} ::= [\cdot] \mid \mathbb{M} \gg\!\!= e \qquad\qquad \mathbb{F} \in FCtxt_{CH} ::= \mathbb{E} \mid (\mathtt{takeMVar}\,\mathbb{E}) \mid (\mathtt{putMVar}\,\mathbb{E}\,e)$$

<div align="center">Figure 4: Syntax of processes, expressions, types, and context classes of CH</div>

Functional Evaluation:

(cpce) $\quad \Leftarrow \mathbb{M}[\mathbb{F}[x]] \mathbin{|} x = e \xrightarrow{sr} \Leftarrow \mathbb{M}[\mathbb{F}[e]] \mathbin{|} x = e$

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

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

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

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

Monadic Computations:

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

(tmvar) $\quad \Leftarrow \mathbb{M}[\mathtt{takeMVar}\ x] \mathbin{|} x\,\mathbf{m}\,e \xrightarrow{sr} \Leftarrow \mathbb{M}[\mathtt{return}\ e] \mathbin{|} x\,\mathbf{m}\,{-}$

(pmvar) $\quad \Leftarrow \mathbb{M}[\mathtt{putMVar}\ x\ e] \mathbin{|} x\,\mathbf{m}\,{-} \xrightarrow{sr} \Leftarrow \mathbb{M}[\mathtt{return}\ ()] \mathbin{|} x\,\mathbf{m}\,e$

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

(fork) $\quad \Leftarrow \mathbb{M}[\mathtt{forkIO}\ e] \xrightarrow{sr} \Leftarrow \mathbb{M}[\mathtt{return}\ ()] \mathbin{|} \Leftarrow e$

Closure w.r.t. $\mathbb{D}$-contexts and $\equiv$: 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 reductions.

<div align="center">Figure 5: Standard reduction rules of CH (call-by-name-version)</div>

3 The Process Calculus *CH*

The calculus *CH* (a variant of the language *CHF*, [23, 24]) models a core language of Concurrent Haskell [20]. We assume a partitioned set of *data constructors* c where each family represents a type T. The data constructors of type T are $c_{T,1},\ldots,c_{T,|T|}$ 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 $\mathtt{Bool}$ with constructors $\mathtt{True}$, $\mathtt{False}$, a type $\mathtt{List}$ with constructors $\mathtt{Nil}$ and $:$ (written infix), and a type $\mathtt{Pair}$ with a constructor $(,)$ written (a,b).

Processes $P \in Proc_{CH}$ in *CH* have expressions $e \in Expr_{CH}$ as subterms. See Fig. 4 where u,w,x,y,z are variables from an infinite set *Var*. Processes are formed by parallel composition " $\mathbin{|}$ ". The ν-binder restricts the scope of a variable. A concurrent thread $\Leftarrow e$ evaluates e. In a process there is (at most one) unique distinguished thread, called *main thread*, written as $\overset{\text{main}}{\Longleftarrow} e$. 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 $x\,\mathbf{m}\,{-}$. Here x is called the *name of the MVar*. Bindings $x = e$ model the global heap, of shared expressions, where x is called a *binding variable*. If x is a name of an MVar or a binding variable, then x is called an *introduced variable*. In $Q \mathbin{|} \nu x.P$ the scope of x is P. A process is *well-formed*, if all introduced variables are pairwise distinct and there exists at most one main thread $\overset{\text{main}}{\Longleftarrow} e$.

Expressions $Expr_{CH}$ consist of functional and monadic expressions $MExpr_{CH}$. Functional expres-

sions are variables, *abstractions* $\lambda x.e$, *applications* $(e_1\ e_2)$, seq-*expressions* (seq $e_1\ e_2$), *constructor applications* $(c\ e_1\ \dots\ e_{\mathrm{ar}(c)})$, letrec-*expressions* (letrec $x_1 = e_1,\dots,x_n = e_n$ in e), and case$_T$-*expressions* for every type T. We abbreviate case-expressions as case$_T\ e$ of *alts* where *alts* are the case-*alternatives* such that there is exactly one alternative $(c_{T,i}\ x_1 \dots x_{\mathrm{ar}(c_{T,i})}\ \text{->}\ e_i)$ 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 often omit the type index T in case$_T$. In letrec $x_1 = e_1,\dots,$ $x_n = e_n$ 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 $\gg=$ implements sequential composition of IO-operations, the forkIO-operator performs thread creation, and return lifts expressions into the monad.

Monadic values are newMVar e, takeMVar e, putMVar $e_1\ e_2$, return e, $e_1 \gg= e_2$, or forkIO e. *Functional values* are abstractions and constructor applications. A *value* is a functional or a monadic value.

Abstractions, letrec-expressions, case-alternatives, and $\nu x.P$ introduce variable binders. This induces bound and free variables (dentoted by $FV(\cdot)$), α-renaming, and α-equivalence $=_\alpha$. If $FV(P) = \emptyset$, then we call process P *closed*. We assume the *distinct variable convention*: free variables are distinct from bound variables, and bound variables are pairwise distinct. We assume that α-renaming is applied to obey this convention. Structural congruence $\equiv$ of *CH*-processes is the least congruence satisfying the laws $P_1 \mid P_2 \equiv P_2 \mid P_1$, $(P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3)$, $\nu x_1.\nu x_2.P \equiv \nu x_2.\nu x_1.P$, $P_1 \equiv P_2$ if $P_1 =_\alpha P_2$, and $(\nu x.P_1) \mid P_2 \equiv \nu x.(P_1 \mid P_2)$ if $x \notin FV(P_2)$.

We assume expressions and processes to be well-typed w.r.t. a monomorphic type system: the typing rules are standard (they can be found in [29]). The syntax of types is in Fig. 4 where $(\text{IO}\ t)$ is the type of a monadic action with return type t, $(\text{MVar}\ t)$ is the type of an MVar with content type t, and $t_1 \rightarrow t_2$ 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 call-by-need semantics (see [23] for the calculus *CHF*). However, the equivalence of the reduction strategies is not important for this paper. That is why we do not include it.

In *CH*, a context is a process or an expression with a (typed) hole $[\cdot]$. We introduce several classes of contexts in Fig. 4. They are used by the reduction rules.

Definition 3.1. *The standard reduction $\xrightarrow{sr}$ is defined by the rules and the closure in Fig. 5. It is only permitted for well-formed processes which are not successful.*

Functional evaluation includes β-reduction (beta), copying shared bindings into needed positions (cpce), evaluating case- and seq-expressions (case) and (seq), and moving letrec-bindings into the global bindings (mkbinds). For monadic computations, rule (lunit) implements the monadic evaluation. 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 thread. A concurrent thread of the form $\Leftarrow$ return e is terminated (where e is of type $()$).

Example 3.2. *The process $\xLeftarrow{\text{main}}$ newMVar $()\gg= (\lambda y.\text{forkIO}\,(\text{takeMVar}\,y)) \gg= \lambda\_.\text{putMVar}\,y\,()$ creates a filled MVar, that is emptied by a spawned thread, and then again filled by the main thread.*

We say that a *CH*-process P is *successful* if $P \equiv \nu x_1 \dots x_n.(\xLeftarrow{\text{main}} \text{return}\ e \mid P')$ and if P is well-formed. This captures Haskell's behavior that termination of the main-thread terminates all threads.

Definition 3.3. *Let P be a CH-process. Then P may-converges (denoted as $P{\downarrow}$), iff P is well-formed and $\exists P' : P \xrightarrow{sr,*} P'$ such that P' is successful. If $P{\downarrow}$ does not hold, then P must-diverges and we write $P{\Uparrow}$. 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 $P{\uparrow}$.*

$$\begin{aligned}
\tau_0(P) &\overset{\text{main}}{=\!\Longleftarrow} \mathbf{do}\ \{stop \leftarrow \text{newMVar}\ ();\text{forkIO}\ \tau(P);\text{putMVar}\ stop\ ()\} \\
\tau(\overline{x}y.P) &= \mathbf{do}\ \{checkx \leftarrow \text{newMVar}\ ();\text{putMVar}\ (unchan\ x)\ (y,checkx);\text{putMVar}\ checkx\ ();\tau(P)\} \\
\tau(x(y).P) &= \mathbf{do}\ \{(y,checkx) \leftarrow \text{takeMVar}\ (unchan\ x);\text{takeMVar}\ checkx;\tau(P)\}\} \\
\tau(P \mid Q) &= \mathbf{do}\ \{\text{forkIO}\ \tau(Q);\tau(P)\} \\
\tau(\nu x.P) &= \mathbf{do}\ \{chanx \leftarrow \text{newEmptyMVar};\text{letrec}\ x = \text{Chan}\ chanx\ \text{in}\ \tau(P)\} \\
\tau(0) &= \text{return}\ () \\
\tau(\text{Stop}) &= \text{takeMVar}\ stop \\
\tau(!P) &= \text{letrec}\ f = \mathbf{do}\ \{\text{forkIO}\ \tau(P);f\}\ \text{in}\ f
\end{aligned}$$

Figure 6: Translations τ_0 and τ

Definition 3.4. *Contextual approximation $\leq_c$ and 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 $\forall \mathbb{D} \in PCtxt_{CH} : \mathbb{D}[P_1]\downarrow \implies \mathbb{D}[P_2]\downarrow$ and $P_1 \leq_{c,\Downarrow} P_2$ 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 $e_1 \leq_c e_2 \wedge e_2 \leq_c e_1$.*

As an example, we consider the processes

$$\begin{aligned}
P_1 &:= \nu m.(\overset{\text{main}}{\Longleftarrow} \text{takeMVar}\ m \mid {\Leftarrow} \text{takeMVar}\ m \mid m\,\mathbf{m}\,()) \\
P_2 &:= \overset{\text{main}}{\Longleftarrow} \text{return}\ () \\
P_3 &:= \overset{\text{main}}{\Longleftarrow} \text{letrec}\ x = x\ \text{in}\ x
\end{aligned}$$

Process P_1 is may-convergent and may-divergent (and thus not should-convergent), since either the main-thread succeeds in emptying the MVar m, or (if the other threads empties the MVar m) the main-thread is blocked forever. The process P_2 is sucessful. The process P_3 is must-divergent. The equivalence $P_1 \sim_{c,\downarrow} P_2$ holds, but $P_1 \not\sim_c P_2$, since P_2 is should-convergent and thus $P_1 \not\sim_{c,\Downarrow} P_2$. As a further example, it is easy to verify that $P_1 \sim_{c,\Downarrow} P_3$ holds, since both processes are not should-convergent and a surrounding context cannot change this. However, $P_1 \not\sim_{c,\downarrow} P_3$, since $P_3 \Uparrow$.

Contextual approximation and equivalence are (pre)-congruences. The following equivalence will help to prove properties of our translation.

Lemma 3.5. *The relations in Definition 3.4 are unchanged, if we add closedness: for $\xi \in \{\downarrow,\Downarrow\}$, let $P_1 \leq_{c,\xi} P_2$ iff $\forall \mathbb{D} \in PCtxt_{CH}$ such that $\mathbb{D}[P_1],\mathbb{D}[P_2]$ are closed: $\mathbb{D}[P_1]\xi \implies \mathbb{D}[P_2]\xi$.*

4   The Translation τ_0 with Private MVars

We present a translation τ_0 that encodes Π_{Stop}-processes as *CH*-programs. It establishes correct synchronous communication by using a private MVar, which is created by the sender and its name is sent to the receiver. The receiver uses it to acknowledge that the message was received. Since only the sender and the receiver know this MVar, no other thread can interfere the communication.The approach has similarities with Boudol's translation [3] from the π-calculus into an asynchronous one, where a private channel name of the π-calculus was used to guarantee safe communication between sender and receiver.

For translating π-calculus channels into *CH*, we use a recursive data type Channel (with constructor Chan), which is defined in Haskell-syntax as

$$\text{data Channel} = \text{Chan}\ (\text{MVar}\ (\text{Channel},(\text{MVar}\ ())))$$

We abbreviate $(\mathtt{case_{Chan}}\ e\ \mathtt{of}\ (\mathtt{Chan}\ m\ \mathtt{->}\ m))$ as *(unchan e)*.

We use $a \gg b$ for $a \gg= (\lambda\_.b)$ and also use Haskell's do-notation with the following meaning:

$$\mathbf{do}\ \{x \leftarrow e_1; e_2\} \quad = e_1 \gg= \lambda x.(\mathbf{do}\ \{e_2\}) \qquad\qquad \mathbf{do}\ \{e_1; e_2\} = e_1 \gg (\mathbf{do}\ \{e_2\})$$
$$\mathbf{do}\ \{(x,y) \leftarrow e_1; e_2\} = e_1 \gg= \lambda z.\mathtt{case_{Pair}}\ z\ \mathtt{of}\ (x,y) \mathtt{->} (\mathbf{do}\ \{e_2\}) \qquad \mathbf{do}\ \{e\} \quad\quad = e$$

As a further abbreviation, we write $y \leftarrow \mathtt{newEmptyMVar}$ inside a **do**-block to abbreviate the sequence $y \leftarrow \mathtt{newMVar}\ \bot; \mathtt{takeMVar}\ y$, where $\bot$ is a must-divergent expression. Our translation uses one MVar per channel which contains a pair consisting of the (translated) name of the channel and a further MVar used for the synchronization, which is private, i.e. only the sender and the receiver know it. Privacy is established by the sender: it creates a new MVar for every send operation. Message y is sent over channel x by sending a pair $(y, check)$ where *check* is an MVar containing (). The receiver waits 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*[3]

Definition 4.1. *We define the translation τ_0 and its inner translation τ from the $\Pi_{\mathtt{Stop}}$-calculus into the CH-calculus in Fig. 6. 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 $\Pi_{\mathtt{Stop}}$-calculus into *CH*-expressions. Except for the main-thread (and using keyword $\mathtt{let}$ instead of $\mathtt{letrec}$), the translation τ_0 generates a valid Concurrent Haskell-program, i.e. if we write $\tau_0(P) = \overset{\mathrm{main}}{\Longleftarrow} e$ as $\mathtt{main} = e$, we can execute the translation in the Haskell-interpreter.

Example 4.2. *We consider the $\Pi_{\mathtt{Stop}}$-process $P := \nu x, y_1, y_2, z.(x(y_1).0 \mathbin{|} x(y_2).\mathtt{Stop} \mathbin{|} \bar{x}z.0)$ which is may-convergent and may-divergent: depending on which receiver communicates with the sender, the result is the successful process $\nu x, y_1.(x(y_1).0 \mathbin{|} \mathtt{Stop})$ or the must-divergent process $\nu x, y_2.(x(y_2).\mathtt{Stop})$. The CH-process $\tau_0(P)$ reduces after several steps to the process*

$$\nu stop, chanx, chany_1, chany_2, chanz, checkx, x, y_1, y_2, z.($$
$$chanx\,\mathbf{m}\,(z, checkx) \mathbin{|} chany_1\,\mathbf{m}- \mathbin{|} chany_2\,\mathbf{m}- \mathbin{|} chanz\,\mathbf{m}- \mathbin{|} checkx\,\mathbf{m}() \mathbin{|} stop\,\mathbf{m}()$$
$$\mathbin{|} x{=}\mathrm{Chan}\,chanx \mathbin{|} z{=}\mathrm{Chan}\,chanz \mathbin{|} y_1{=}\mathrm{Chan}\,chany_1 \mathbin{|} y_2{=}\mathrm{Chan}\,chany_2 \mathbin{|} \overset{\mathrm{main}}{\Longleftarrow} \mathtt{putMVar}\ stop\ ()$$
$$\mathbin{|} \Leftarrow \mathbf{do}\ \{\mathtt{putMVar}\ checkx\ (); \mathtt{return}\ ()\}$$
$$\mathbin{|} \Leftarrow \mathbf{do}\ \{(y_1, checkx) \leftarrow \mathtt{takeMVar}\ chanx; \mathtt{takeMVar}\ checkx; \mathtt{return}\ ()\}$$
$$\mathbin{|} \Leftarrow \mathbf{do}\ \{(y_2, checkx) \leftarrow \mathtt{takeMVar}\ chanx; \mathtt{takeMVar}\ checkx; \mathtt{takeMVar}\ stop\})$$

Now the first thread (which is the translation of sender $\bar{x}z.0$) is blocked, since it tries to fill the full MVar checkx. The second thread (the encoding of $x(y_1).0$) and the third thread (the encoding of $x(y_2).\mathtt{Stop}$) race for emptying the MVar chanx. If the second thread wins, then it will fill the MVar checkx which is then emptied by the first thread, and all other threads are blocked forever. If the third thread wins, then it will fill the MVar checkx which is then emptied by the first thread, and then the second thread will empty the MVar stop. This allows the main-thread to fill it, resulting in a successful process.

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

---

[3]A variant of the translation would be to change the roles for the acknowledgement such that an empty MVar is created, which is filled by the receiver and emptied by the sender. The reasoning on the correctness of the translation is very similar to the one presented here.

Definition 4.3. *Let* $C_{out}^{\tau} := \nu stop. \xleftarrow{\text{main}} \mathbf{do}\ \{stop \leftarrow \texttt{newMVar}\ ();\texttt{forkIO}\ [\cdot];\texttt{putMVar}\ stop\ ()\}$. *Variants* $\downarrow_0, \Downarrow_0$ *of may- and should-convergence of expressions* e *within the context* C_{out}^{τ} *in CH are defined as* $e \downarrow_0$ *iff* $C_{out}^{\tau}[e] \downarrow$ *and* $e \Downarrow_0$ *iff* $C_{out}^{\tau}[e] \Downarrow$. *The relation* $\sim_{c,\tau_0}$ *is defined by* $\sim_{c,\tau_0} := \leq_{c,\tau_0} \cap \geq_{c,\tau_0}$, *where* $e_1 \leq_{c,\tau_0} e_2$ *iff* $\forall C : if\ FV(C[e_1], C[e_2]) \subseteq \{stop\}$, *then* $C[e_1] \downarrow_0 \implies C[e_2] \downarrow_0$ *and* $C[e_1] \Downarrow_0 \implies C[e_2] \Downarrow_0$.

Since $\leq_{c,CH}$ is a subset of $\leq_{c,\tau_0}$, we often can use the more general relations for reasoning.

Definition 4.4. *Let* $\Pi_{\texttt{Stop},C}$ *be the contexts of* $\Pi_{\texttt{Stop}}$. *We define the following properties for* τ_0 *and* τ *(see [31] for a general framework of properties of translations under observational semantics). For open processes* P, P', *we say that translation* τ *is*

convergence-equivalent *iff for all* $P \in \Pi_{\texttt{Stop}}$: $P \downarrow \iff \tau(P) \downarrow_0$ *and* $P \Downarrow \iff \tau(P) \Downarrow_0$,

compositional upto $\{\downarrow_0, \Downarrow_0\}$ *iff for all* $P \in \Pi_{\texttt{Stop}}$, *all* $C \in \Pi_{\texttt{Stop},C}$, *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* $P, P' \in \Pi_{\texttt{Stop}}$: $\tau(P) \leq_{c,\tau_0} \tau(P') \implies P \leq_c P'$, *and*

fully abstract *iff for all processes* $P, P' \in \Pi_{\texttt{Stop}}$: $P \leq_c P' \iff \tau(P) \leq_{c,\tau_0} \tau(P')$.

Convergence-equivalence of translation τ_0 for may- and should-convergence holds. For readability the proof is omitted, but given in the technical report [29], where we show:

Theorem 4.5. *Let* $P \in \Pi_{\texttt{Stop}}$ *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.* $P \downarrow \iff \tau(P) \downarrow_0$ *and* $P \Downarrow \iff \tau(P) \Downarrow_0$.

We show that the translation is adequate (see Theorem 4.8 below). The interpretation of this result is that the π-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 π-calculus does not have a notion of values, 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 (see Theorem 4.9 below), which is rather natural, since it only means that it is mapped into a subset of the *CH*-expressions and that this is a proper subset w.r.t. the semantics. For proving both theorems, we first use a simple form of a context lemma:

Lemma 4.6. *Let* e, e' *be CH-expressions, where the only free variable in* e, e' *is stop.*
    Then $C_{out}^{\tau}[e] \leq_c C_{out}^{\tau}[e']$ *holds, if and only if* $C_{out}^{\tau}[e] \downarrow \implies C_{out}^{\tau}[e'] \downarrow$ *and* $C_{out}^{\tau}[e] \Downarrow \implies C_{out}^{\tau}[e'] \Downarrow$.

Proposition 4.7. *The translation* τ *is compositional upto* $\{\downarrow_0, \Downarrow_0\}$.

We show that the translation τ transports $\Pi_{\texttt{Stop}}$-processes into *CH*, such that adequacy holds. Thus the translated processes also correctly mimic the behavior of the original $\Pi_{\texttt{Stop}}$-processes when plugged into contexts. If the translated open processes cannot be distinguished by $\leq_{c,\tau_0}$, i.e. there is no test that detects a difference w.r.t. may- and should-convergence, then the original processes are equivalent in the π-calculus. However, this open translation is not fully abstract, which means that there are *CH*-contexts (not in the image of the translation) that can see and exploit too much of the details of the translation.

Theorem 4.8. *The translation* τ *is adequate.*

Proof. We prove the adequacy for the preorder $\leq_{c,\tau_0}$, for $\sim_{c,\tau_0}$ and $\sim_c$ the claim follows by symmetry. Let P, P' be $\Pi_{\texttt{Stop}}$-processes, such that $\tau(P) \leq_{c,\tau_0} \tau(P')$. We show that $P \leq_c P'$. We use Lemma 3.5 to restrict considerations to closed $C[P], C[P']$ below. Let C be a context in $\Pi_{\texttt{Stop}}$, such that $C[P], C[P']$ are closed and $C[P] \downarrow$. Then $\tau_0(C[P]) = C_{out}^{\tau}[\tau(C[P])]$. Closed convergence-equivalence implies $C_{out}^{\tau}[\tau(C[P])] \downarrow$. By Proposition 4.7. we have $C_{out}^{\tau}[\tau(C)[\tau(P)]] \downarrow$. Now $\tau(P) \leq_{c,\tau_0} \tau(P')$ implies $C_{out}^{\tau}[\tau(C)[\tau(P')]] \downarrow$, which is the same as $C_{out}^{\tau}[\tau(C[P'])] \downarrow$ using Proposition 4.7. Closed convergence-equivalence implies $C[P'] \downarrow$. The same arguments hold for $\Downarrow$ instead of $\downarrow$. In summary, we obtain $P \leq_c P'$. □

Theorem 4.9. *The translation τ is not fully abstract, but it is fully abstract on closed processes, i.e. for closed processes $P_1, P_2 \in \Pi_{\texttt{Stop}}$, we have $P_1 \leq_c P_2 \iff \tau(P_1) \leq_{c,\tau_0} \tau(P_2)$.*

Proof. The first part holds, since an open translation can be closed by a context without initializing the ν-bound MVars. For $P = \bar{x}(y).\texttt{Stop} \mid x(z).\texttt{Stop}$, we have $P \sim_c \texttt{Stop}$ but $\tau(P) \not\sim_{c,0} \tau(\texttt{Stop})$: let $\mathbb{D}$ be a context that does not initialize the MVars for x (as the translation does). Then $\mathbb{D}[\tau(P)]\Uparrow_0$, but $\mathbb{D}[\tau(\texttt{Stop})]\Downarrow_0$. Restricted to closed processes, full abstraction holds: $P_1 \leq_c P_2 \Longrightarrow \tau(P_1) \leq_{c,\tau_0} \tau(P_2)$ follows from Lemma 4.6, since τ_0 produces closed processes in context C_{out}^τ. Theorem 4.8 implies the other direction. $\qquad\square$

5   Translations with Global MVars

In this section we investigate translations that do not use private MVars, but use a fixed number of global MVars. We first motivate this investigation. The translation τ is quite complex and thus we want to figure out whether there are simpler translations. A further reason is that τ is not optimal, since it generates one MVar per communication which can be garbage-collected after the communication, however, generation and garbage collection require resources and thus the translation τ may be inefficient in practice.

To systematically search for small global translations we implemented an automated tool. It searches for translations with global MVars (abstracting from a lot of other aspects of the translation) and tries to refute the correctness. As we show, most of the small translations are shown as incorrect by our tool. Analyzing correctness of the remaining translations can then be done by hand.

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 $\texttt{Stop}$-constant) are not discussed and we assume that they are encoded as before (as the translation τ did). We also keep the main idea to translate a channel of the π-calculus into *CH* by representing it as an object of a user-defined data type $\texttt{Channel}$ that consists of an MVar for transferring the message (which again is a $\texttt{Channel}$), and additional MVars for implementing a correct synchronization mechanism. For the translation τ, we used a private MVar (created by the sender, and transferred together with the message). Now 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 we call *check-MVars*) to be of type $\texttt{MVar ()}$. Such MVars are comparable to binary semaphores, where filling and emptying correspond to operations signal and wait. In summary, we analyze translations of π-calculus channels into a *CH*-data type $\texttt{Channel}$ defined in Haskell-syntax as

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

A π-calculus channel x is represented as a *CH*-binding $x = \texttt{Chan}\ content\ check_1\ \ldots\ check_n$ where $content, check_1, \ldots, check_n$ are appropriately initialized (i.e. empty) MVars. The MVars are public (or global), since all processes which know x have access to the components of the channel. After fixing this representation of a π-channel in *CH*, the task is to translate the input- and output-actions $x(y)$ and $\bar{x}z$ into *CH*-programs such that the interaction reduction is performed correctly and synchronously. We call the translation of $x(y)$, the *receiver* (program) and the translation of $\bar{x}z$ the *sender* (program). As a simplification, we restrict the allowed operations of the sender and receiver allowing only the operations:

$$\phi_{0,T}(P) \quad \overset{\text{main}}{=\!=\!=} \textbf{do } \{stop \leftarrow \texttt{newMVar } ();\texttt{forkIO } \phi_T(P);\texttt{putMVar } stop \, ()\}$$

$$\phi_T(\overline{x}y.P) \quad =\textbf{do } \{T_{send}^{x,y}; \phi_T(P)\}$$

$$\phi_T(x(y).P) =\textbf{do } \{T_{receive}^{x,y}; \phi_T(P)\}$$

$$\phi_T(P \mid Q) \quad =\textbf{do } \{\texttt{forkIO } \phi_T(Q); \phi_T(P)\}$$

$$\phi_T(\nu x.P) \quad =\textbf{do}\{contx \leftarrow \texttt{newEmptyMVar}; checkx_1 \leftarrow \texttt{newEmptyMVar}; \ldots; checkx_n \leftarrow \texttt{newEmptyMVar};$$
$$\texttt{letrec } x = \texttt{Chan } contx \, checkx_1 \ldots checkx_n \texttt{ in } \phi_T(P)\}$$

$$\phi_T(0) \quad =\texttt{return } ()$$

$$\phi_T(\texttt{Stop}) =\texttt{takeMVar } stop$$

$$\phi_T(!P) \quad =\texttt{letrec } f = \textbf{do } \{\texttt{forkIO } \phi_T(P); f\} \texttt{ in } f$$

Figure 7: Induced translations ϕ_T and $\phi_{0,T}$ where $T = (T_{send}, T_{recieve})$ uses n check-MVars

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

takeS: The receiver takes the message from the *contents*-MVar of channel x and replaces name y by the received name in the subsequent program. The operation occurs exactly once in the receiver program. We write it as $\texttt{takeS}_x \, y$, or as $\texttt{takeS}$, if x and y are clear. It represents the *CH*-expression $\texttt{case}_{\texttt{Channel}} \, x \texttt{ of } (\texttt{Chan } c \, a_1 \ \ldots \ a_n \texttt{ -> takeMVar } c \ggg \lambda y.e)$ where e is the remaining program of the receiver. In **do**-notation, we write $\textbf{do } \{y \leftarrow \texttt{takeS}_x; e\}$ to abbreviate the above *CH*-expression.

putC **and** takeC: The sender and the receiver may synchronize on a check-MVar $check_i$ by putting $()$ into it or by emptying the MVar. These operations are written as $\texttt{putC}_x^i$ and $\texttt{takeC}_x^i$, or also as $\texttt{putC}^i, \texttt{takeC}^i$ if the name x is clear. We write $\texttt{putC}$ and $\texttt{takeC}$ 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*-expression $\texttt{case}_{\texttt{Channel}} \, x \texttt{ of } (\texttt{Chan } c \, a_1 \ \ldots \ a_n \texttt{ -> putMVar } a_i \, () \gg e)$ and $\texttt{takeC}_x^i$ represents the expression $\texttt{case}_{\texttt{Channel}} \, x \texttt{ of } (\texttt{Chan } c \, a_1 \ \ldots \ a_n \texttt{ -> takeMVar } a_i \gg e)$.

We restrict our search for translations to the case that the sender and the receiver programs are sequences of the above operations, assuming that they are independent of the channel name x. With this restriction, we can abstractly write the translation of the sender and the receiver 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:

Definition 5.1. *Let $n > 0$ be a number of check-MVars. A* standard global synchronized-to-buffer transla-tion *(or* gstb-translation*) is a pair $(T_{send}, T_{receive})$ of a send-sequence T_{send} and a receive-sequence $T_{receive}$ consisting of $\texttt{putS}, \texttt{takeS}, \texttt{putC}^i$ and $\texttt{takeC}^i$ operations, where the send-sequence contains $\texttt{putS}$ once, the receive-sequence contains $\texttt{takeS}$ once, and for every $\texttt{putC}^i$-action in $(T_{send}, T_{receive})$, there is also a $\texttt{takeC}^i$-action in $(T_{send}, T_{receive})$. W.l.o.g., we 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 $i < j$ holds.*

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} instan-tiated for $\overline{x}y$, i.e. $\texttt{putS}$ is $\texttt{putS}_x \, y$, and all other operations are indexed with x. We write $T_{receive}^{x,y}$ for the program $T_{receive}$ instantiated for $x(y)$, i.e. $\texttt{takeS}$ is $\texttt{takeS}_x \, y$, and all other operations are indexed with x. The induced translations $\phi_{0,T}$ and ϕ_T of $(T_{send}, T_{receive})$ are defined in Fig. 7.*

The induced translations are defined similar to the translations τ_0 and τ, where the differences are the representations of the channel. The translation of νx, $x(y)$, and $\bar{x}y$ is different, but the other cases remain the same. Since $\phi_{0,T}(P) = C_{out}^\tau[\phi_T(P)]$ and by the same arguments as in Theorem 4.8, we have:

Proposition 5.3. *If ϕ_T is closed convergence-equivalent, then ϕ_T is adequate.*

An *execution* of a translation $(T_{send}, T_{receive})$ for name x is 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.

To speak about the translations we make further classifications: We say that a translation allows *multiple uses*, if a check-MVar is used more than once, i.e. the sender or receiver may contain $\texttt{takeC}^i$ or $\texttt{putC}^i$ more than once for the same i. A translation has the *interprocess check restriction*, if for every i: $\texttt{takeC}^i$ and $\texttt{putC}^i$ do not occur both in T_{send}, and also not both in $T_{receive}$.

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

- executable *if there is a deadlock free execution of T;*

- communicating, *if T_{send} contains at least one $\texttt{takeC}^i$-action;*

- overlap-free *if for a fixed name x, starting with empty MVars, every interleaved (concurrent) execution of $(T_{send}, T_{receive})$ cannot be disturbed by starting another execution of T_{send} and/or $T_{receive}$. More formally, let $((s_1; \ldots; s_i); (r_1; \ldots, r_j))$ and $((s'_1; \ldots; s'_i); (r'_1; \ldots, r'_j))$ 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*

  1. *One part contains only s- and r-actions, and the other part contains only s'- and r'-actions.*
  2. *One part contains only s- and r'-actions and the other part contains only s'- and r-actions.*

We implemented a tool to enumerate translations and to test whether each translation preserves and reflects may- and should-convergence for a (given) finite set of processes. Hence, our tool can refute the correctness of translations, but it can also output (usually small) sets of translations which are not refuted and which are promising candidates for correct translations. 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 $\nu x.(x(y) \mid \bar{x}z)$ (which is must-divergent). Thus, we apply the translation to a subset of π-processes, which we view as critical and for which we can automatically decide may- and should-convergence (before and after the translation). We consider only π-programs of the form $(\nu x_1, \ldots, x_n.P)$ where P contains only 0, $\texttt{Stop}$, parallel composition, and input- and output-prefixes. These 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. We also implemented techniques to generates all such programs until a bound on the size of the program is reached.

Our simulation tool[4] can execute all possible evaluations of those π-processes and – since all evaluation paths are finite – 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, but generate a sequence of sequences of $\texttt{takeS}_x, \texttt{putS}_x, \texttt{takeC}_x^i, \texttt{putC}_x^i z$ and $\texttt{Stop}$-operations by applying the translation to all action prefixes in the π-program and by encoding $\texttt{Stop}$ as $\texttt{Stop}, 0$ into an empty sequence. We get a sequence of sequences, since we have several threads and each thread is represented by one sequence. For executing the translated program, we simulate the global store (of MVars) and execute all possible interleavings

---

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

where we check for may- and should-convergence by looking whether the `Stop` eventually occurs at the beginning of the sequence. This simulates the behavior of the real *CH*-program in a controllable manner.

With the encoding of the sender- and receiver program and a π-calculus process P we

1. translate P with the encodings in the sequence of sequences consisting of $\mathtt{takeS}_x$, $\mathtt{putS}_x$, $\mathtt{takeC}_x^i$, $\mathtt{putC}_x^i\ z$ and `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.

Example 5.5. *Let us consider the gstb-translation* $(T_{send}, T_{receive}) = ([\mathtt{takeC}, \mathtt{putS}], [\mathtt{putC}, \mathtt{takeS}])$ *and the π-process* $P = \nu x, y, z, w(\bar{x}y.x(z).\mathtt{Stop} \mid x(w).0)$. *Our tool recognizes that* $P\!\uparrow$ *and* $P\!\Uparrow$ *holds, since* P *reduces to the must-divergent process* $\nu x, z.(x(z).\mathtt{Stop})$ *and there are no other reduction possibilities.*

Applying $(T_{send}, T_{receive})$ *to* P *yields the abstract program*

$$q := [[\mathtt{takeC}_x, \mathtt{putS}_x\ y, \mathtt{putC}_x, \mathtt{takeS}_x\ z, \mathtt{Stop}], [\mathtt{putC}_x, \mathtt{takeS}_x\ w]].$$

For q, our tool inspects all executions. Among them there is the sequence

$$\mathtt{putC}_x; \mathtt{takeC}_x; \mathtt{putS}_x\ y; \mathtt{putC}_x; \mathtt{takeS}_x\ z; \mathtt{Stop}$$

which can be executed ending in `Stop`. *Thus q is may-convergent, and thus the process P is a counter-example that refutes the correctness of the translation.*

The case that there is no check-MVar leads to one possible translation $([\mathtt{putS}], [\mathtt{takeS}])$ which means that $\bar{x}z$ 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 $\bar{x}z.x(y).\mathtt{Stop}$ is neither may- nor should-convergent, but the translation (written as an abstract program) is $[[\mathtt{putS}_x\ z, \mathtt{takeS}_x\ y, \mathtt{Stop}]]$. I.e., it consists of one process which is may- and should-convergent (since $\mathtt{putS}_x\ z; \mathtt{takeS}_x\ y; \mathtt{Stop}$ is the only evaluation sequence and its execution ends in `Stop`). Note that the translation into *CH* will generate two threads: the main threads that will wait until the MVar *stop* is filled, and a concurrent thread that will do the above operations.

5.1 Translations with Interprocess Check Restriction

We consider translations with the interprocess check restriction (each $\mathtt{takeC}^i$ and $\mathtt{putC}^i$ must be distributed between the sender and the receiver). There are $n! \cdot 2^n \cdot (n+1)^2$ different translations (for n check-MVars). For a single check-MVar, all 8 translations are rejected by our tool, Table 1 shows the translations and the obtained counter examples. For 2 check-MVars, our tool refutes all 72 translations. Compared to Table 1, there are two further π-programs used as counterexample. However, also the programs $\bar{x}y.\mathtt{Stop} \mid x(y)$ and $\bar{x}y.x(z).\bar{z}q \mid x(z) \mid x(z) \mid \bar{x}z \mid y(u).\mathtt{Stop}$ suffice to refute all 72 translations.

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

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

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

Translation (sender,receiver)	Counter-example (π-process)	↓ before	⇓ before	↓ after	⇓ after
$([\text{putC},\text{putS}],[\text{takeC},\text{takeS}])$	$\bar{x}y.x(y).\text{Stop}$	N	N	Y	Y
$([\text{putC},\text{putS}],[\text{takeS},\text{takeC}])$	$\bar{x}y.x(y).\text{Stop}$	N	N	Y	Y
$([\text{putS},\text{putC}],[\text{takeC},\text{takeS}])$	$\bar{x}y.x(y).\text{Stop}$	N	N	Y	Y
$([\text{putS},\text{putC}],[\text{takeS},\text{takeC}])$	$\bar{x}y.x(y).\text{Stop}$	N	N	Y	Y
$([\text{takeC},\text{putS}],[\text{putC},\text{takeS}])$	$\bar{x}y.x(z).\text{Stop} \mid x(w)$	N	N	Y	N
$([\text{takeC},\text{putS}],[\text{takeS},\text{putC}])$	$\bar{x}y.\text{Stop} \mid x(y)$	Y	Y	N	N
$([\text{putS},\text{takeC}],[\text{putC},\text{takeS}])$	$\bar{x}y.x(z).\text{Stop} \mid x(w)$	N	N	Y	N
$([\text{putS},\text{takeC}],[\text{takeS},\text{putC}])$	$\bar{x}z.\bar{z}a.\text{Stop} \mid \bar{x}w.\bar{w}a.\text{Stop} \mid x(y).y(u)$	Y	Y	Y	N

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

Proof. For $n = 1$, we check the translations in Table 1. The first four are non-communicating. For the translation $([\text{takeC},\text{putS}],[\text{takeS},\text{putC}])$ a deadlock occurs. For $([\text{takeC},\text{putS}],[\text{putC},\text{takeS}])$, after putC, takeC, we can execute putC again. For $([\text{putS},\text{takeC}],[\text{takeS},\text{putC}])$, after executing putS, takeS we can execute putS again. For $([\text{putS},\text{takeC}],[\text{putC},\text{takeS}])$, after putC,putS, takeC we can execute putC again. For $n = 2$, the simulator finds no executable, communicating, and overlap-free translation: 18 translations are non-communicating, 21 lead to a deadlock, and 33 may lead to an overlap. □

For 3 MVars, our tool rejects 762 out of 768 translations (using the same counter examples as for 2 check-MVars) and the following 6 translations remain:

$$\mathfrak{T}_1 = ([\text{putS},\text{putC}^1,\text{takeC}^2,\text{putC}^3],[\text{takeC}^1,\text{putC}^2,\text{takeC}^3,\text{takeS}])$$
$$\mathfrak{T}_2 = ([\text{takeC}^1,\text{putS},\text{takeC}^2,\text{takeC}^3],[\text{putC}^3,\text{putC}^1,\text{takeS},\text{putC}^2])$$
$$\mathfrak{T}_3 = ([\text{putC}^1,\text{putS},\text{takeC}^2,\text{putC}^3],[\text{takeS},\text{putC}^2,\text{takeC}^3,\text{takeC}^1])$$
$$\mathfrak{T}_4 = ([\text{putC}^1,\text{putC}^2,\text{takeC}^3,\text{putS}],[\text{takeC}^2,\text{putC}^3,\text{takeS},\text{takeC}^1])$$
$$\mathfrak{T}_5 = ([\text{takeC}^1,\text{putS},\text{takeC}^2,\text{takeC}^3],[\text{putC}^1,\text{putC}^2,\text{takeS},\text{putC}^3])$$
$$\mathfrak{T}_6 = ([\text{putC}^1,\text{takeC}^2,\text{putS},\text{takeC}^3],[\text{takeC}^1,\text{putC}^2,\text{takeS},\text{putC}^3])$$

Proposition 5.8. *The translations $\mathfrak{T}_1,\mathfrak{T}_2,\mathfrak{T}_3$, and $\mathfrak{T}_4$ are executable, communicating, and overlap-free, whereas the translations $\mathfrak{T}_5$ and $\mathfrak{T}_6$ are executable, communicating, but overlapping.*

Proof. We only consider overlaps. For $\mathfrak{T}_1$ - $\mathfrak{T}_4$, only if all 8 actions are finished, the next send or receive can start. For $\mathfrak{T}_5,\mathfrak{T}_6$, after executing putC^1, takeC^1, we can again execute putC^1. □

In [29] we argue that the induced translation $\phi_{\mathfrak{T}_1}$ 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 then shows:

Theorem 5.9. *Translation $\phi_{\mathfrak{T}_1}$ is adequate.*

For 4 MVars, our tool refutes 9266 and there remain 334 candidates for correct translations.

5.2 Dropping the Interprocess Check Restriction

We now consider gstb-translations without the interprocess check restriction, i.e. putC^i and takeC^i both may occur in the sender-program (or the receiver program, resp.). If we allow one check-MVar without reuse, then there are 20 candidates for translations. All are 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, leading to 420420 candidates for a correct translation – our simulation refutes all of them.

Conjecture 5.10. *We conjecture that there is no correct translation for one check-MVar where re-uses are permitted and the interprocess check restriction is dropped, i.e., T_{send} is a word over* {putS, putC, takeC} *and $T_{receive}$ a word over* {takeS, putC, takeC}, *where* putS, takeS *occur exactly once.* [5]

For two MVars, one use and without the interprocess check restriction there are 420 translations. Our tool refutes all except for two: $\mathfrak{T}_7 = ([\text{putC}^1, \text{putS}, \text{takeC}^2, \text{takeC}^1], [\text{takeS}, \text{putC}^2])$ and $\mathfrak{T}_8 = ([\text{takeC}^1, \text{putS}], [\text{putC}^2, \text{putC}^1, \text{takeS}, \text{takeC}^2])$. In $\mathfrak{T}_7$ the second check-MVar is used as a mutex for the senders, ensuring that only one sender can operate at a time. $\mathfrak{T}_8$ does the same on the receiver side.

Proposition 5.11. *The translations $\mathfrak{T}_7, \mathfrak{T}_8$ are executable, communicating, overlap-free.*

Proof. The translations are executable and communicating. For $\mathfrak{T}_7$, $\text{putC}^1, \text{putS}$ and takeS are performed in this order. An additional sender cannot execute its first command before the original sender performs takeC^1 and this again is only possible after the receiver program is finished. An additional receiver can only be executed after a putS is performed, which cannot be done by the current sender and receivers. For $\mathfrak{T}_8$, $\text{putC}^2, \text{putC}^1$ and takeC^1 are performed in this order. An additional receiver can only start after 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 putC^1 has been executed again, but the current sender and receiver cannot execute this command. $\qquad\square$

The induced translation $\phi_{\mathfrak{T}_7}$ is (closed) convergence-equivalent [29]. With Proposition 5.3 this shows:

Theorem 5.12. *Translation $\phi_{\mathfrak{T}_7}$ is adequate.*

We are convinced that the same holds for $\mathfrak{T}_8$. 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 are potentially correct. One is $([\text{putC}^1, \text{putS}, \text{takeC}^2, \text{takeC}^1], [\text{putC}^3, \text{takeS}, \text{putC}^2, \text{takeC}^3])$ 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 send an acknowledgement. For 4 MVars, there are 277200 translations and 273210 are refuted, i.e. 3990 are potentially correct.

6   Discussion and Conclusion

We investigated translating the π-calculus into *CH* and showed correctness and adequacy of a translation τ_0 with private MVars for every translated communication. For translations with global names, we started an investigation on exhibiting (potentially) correct translations. We identified several minimal potentially correct translations and characterized all incorrect "small" translations. For two particular global translations, we have shown that they are convergence-equivalent and we proved their adequacy on open processes. The exact form of the translations were found by our tool to search for translations and to refute their correctness. The tool showed that there is no correct gstb-translation with the interprocess check restriction for less than 3 check-MVars. We also may consider extended variants of the π-calculus. We are convinced that adding recursion and sums can easily be built into the translation, while it might be challenging to encode 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 a target language that supports 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.

---

[5] We already have a proof in the meantime, not yet published.

References

[1] Martín Abadi & Andrew D. Gordon (1997): *A Calculus for Cryptographic Protocols: The Spi Calculus*. In: *CCS 1997*, ACM, pp. 36–47, doi:10.1145/266420.266432.

[2] Richard Banach, J. Balázs & George A. Papadopoulos (1995): *A Translation of the Pi-Calculus Into MON-STR*. *J.UCS* 1(6), pp. 339–398, doi:10.3217/jucs-001-06-0339.

[3] Gérard Boudol (1992): *Asynchrony and the Pi-calculus*. Technical Report Research Report RR-1702,inria-00076939, INRIA, France. Available at `https://hal.inria.fr/inria-00076939`.

[4] Mauricio Cano, Jaime Arias & Jorge A. Pérez (2017): *Session-Based Concurrency, Reactively*. In: *FORTE 2017*, *LNCS* 10321, Springer, pp. 74–91, doi:10.1007/978-3-319-60225-7_6.

[5] Avik Chaudhuri (2009): *A concurrent ML library in concurrent Haskell*. In: *ICFP 2009*, ACM, pp. 269–280, doi:10.1145/1596550.1596589.

[6] Koen Claessen & David Sands (1999): *Observable Sharing for Functional Circuit Description*. In: *ASIAN 1999*, *LNCS* 1742, Springer, pp. 62–73, doi:10.1007/3-540-46674-6_7.

[7] Cédric Fournet & Georges Gonthier (2002): *The Join Calculus: A Language for Distributed Mobile Programming*. In: *APPSEM 2000*, *LNCS* 2395, Springer, pp. 268–332, doi:10.1007/3-540-45699-6_6.

[8] Andy Gill (2009): *Type-safe observable sharing in Haskell*. In: *Haskell 2009*, ACM, pp. 117–128, doi:10.1145/1596638.1596653.

[9] Rob van Glabbeek, Ursula Goltz, Christopher Lippert & Stephan Mennicke (2019): *Stronger Validity Criteria for Encoding Synchrony*. In: *The Art of Modelling Computational Systems: A Journey from Logic and Concurrency to Security and Privacy - Essays Dedicated to Catuscia Palamidessi on the Occasion of Her 60th Birthday*, *LNCS* 11760, Springer, pp. 182–205, doi:10.1007/978-3-030-31175-9_11.

[10] Rob J. van Glabbeek (2018): *On the validity of encodings of the synchronous in the asynchronous π-calculus*. *Inf. Process. Lett.* 137, pp. 17–25, doi:10.1016/j.ipl.2018.04.015.

[11] Daniele Gorla (2010): *Towards a unified approach to encodability and separation results for process calculi*. *Inf. Comput.* 208(9), pp. 1031–1053, doi:10.1016/j.ic.2010.05.002.

[12] Kohei Honda & Mario Tokoro (1991): *An Object Calculus for Asynchronous Communication*. In: *ECOOP 1991*, Springer-Verlag, pp. 133–147, doi:10.1007/BFb0057019.

[13] Cosimo Laneve (1996): *On testing equivalence: May and Must Testing in the Join-Calculus*. Technical Report UBLCS 96-04, University of Bologna. Available at `https://www.cs.unibo.it/~laneve/papers/laneve96may.pdf`.

[14] Robin Milner (1999): *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press.

[15] Robin Milner, Joachim Parrow & David Walker (1992): *A Calculus of Mobile Processes, I & II*. *Inform. and Comput.* 100(1), pp. 1–77, doi:10.1016/0890-5401(92)90008-4.

[16] Joachim Niehren, David Sabel, Manfred Schmidt-Schauß & Jan Schwinghammer (2007): *Observational Semantics for a Concurrent Lambda Calculus with Reference Cells and Futures*. *Electron. Notes Theor. Comput. Sci.* 173, pp. 313–337, doi:10.1016/j.entcs.2007.02.041.

[17] Dominic A. Orchard & Nobuko Yoshida (2016): *Effects as sessions, sessions as effects*. In: *POPL 2016*, ACM, pp. 568–581, doi:10.1145/2837614.2837634.

[18] Catuscia Palamidessi (1997): *Comparing the Expressive Power of the Synchronous and the Asynchronous pi-calculus*. In: *POPL 1997*, ACM Press, pp. 256–265, doi:10.1145/263699.263731.

[19] Catuscia Palamidessi (2003): *Comparing The Expressive Power Of The Synchronous And Asynchronous Pi-Calculi. Math. Structures Comput. Sci.* 13(5), pp. 685–719, doi:10.1017/S0960129503004043.

[20] Simon L. Peyton Jones, Andrew Gordon & Sigbjorn Finne (1996): *Concurrent Haskell.* In: *POPL 1996*, ACM, pp. 295–308, doi:10.1145/237721.237794.

[21] Corrado Priami (1995): *Stochastic pi-Calculus. Comput. J.* 38(7), pp. 578–589, doi:10.1093/comjnl/38.7.578.

[22] George Russell (2001): *Events in Haskell, and How to Implement Them.* In: *ICFP 2001*, ACM, pp. 157–168, doi:10.1145/507635.507655.

[23] David Sabel & Manfred Schmidt-Schauß (2011): *A contextual semantics for Concurrent Haskell with futures.* In: *PPDP 2011*, ACM, pp. 101–112, doi:10.1145/2003476.2003492.

[24] David Sabel & Manfred Schmidt-Schauß (2012): *Conservative Concurrency in Haskell.* In: *LICS 2012*, IEEE, pp. 561–570, doi:10.1109/LICS.2012.66.

[25] David Sabel & Manfred Schmidt-Schauß (2015): *Observing Success in the Pi-Calculus.* In: *WPTE 2015*, *OASICS* 46, pp. 31–46, doi:10.4230/OASIcs.WPTE.2015.31.

[26] Davide Sangiorgi & David Walker (2001): *On Barbed Equivalences in pi-Calculus.* In: *CONCUR 200*, *LNCS* 2154, Springer, pp. 292–304, doi:10.1007/3-540-44685-0_20.

[27] Davide Sangiorgi & David Walker (2001): *The π-calculus: a theory of mobile processes.* Cambridge university press.

[28] Manfred Schmidt-Schauß, Joachim Niehren, Jan Schwinghammer & David Sabel (2008): *Adequacy of Compositional Translations for Observational Semantics.* In: *IFIP TCS 2008*, *IFIP* 273, Springer, pp. 521–535, doi:10.1007/978-0-387-09680-3_35.

[29] Manfred Schmidt-Schauß & David Sabel (2020): *Embedding the Pi-Calculus into a Concurrent Functional Programming Language.* Frank report 60, Institut für Informatik. Fachbereich Informatik und Mathematik. J. W. Goethe-Universität Frankfurt am Main. Available at `http://www.ki.informatik.uni-frankfurt.de/papers/frank/frank-60v5.pdf`.

[30] Manfred Schmidt-Schauß, David Sabel & Nils Dallmeyer (2018): *Sequential and Parallel Improvements in a Concurrent Functional Programming Language.* In: *PPDP 2018*, ACM, pp. 20:1–20:13, doi:10.1145/3236950.3236952.

[31] Manfred Schmidt-Schauß, David Sabel, Joachim Niehren & Jan Schwinghammer (2015): *Observational program calculi and the correctness of translations.* *Theor. Comput. Sci.* 577, pp. 98–124, doi:10.1016/j.tcs.2015.02.027.

[32] Jan Schwinghammer, David Sabel, Manfred Schmidt-Schauß & Joachim Niehren (2009): *Correctly translating concurrency primitives.* In: *ML 2009*, ACM, pp. 27–38, doi:10.1145/1596627.1596633.

[33] Ping Yang, C. R. Ramakrishnan & Scott A. Smolka (2004): *A logical encoding of the pi-calculus: model checking mobile processes using tabled resolution.* *STTT* 6(1), pp. 38–66, doi:10.1007/s10009-003-0136-3.