Correctly Implementing Synchronous Message Passing in the Pi-Calculus by Concurrent Haskell’s MVars

Manfred Schmidt-Schauß
Goethe-University Frankfurt

David Sabel
LMU Munich

EXPRESS/SOS 2020
August 31, 2020
General Motivation

- We are interested in the **correctness of translations** between programming languages

- **Questions:**
  - can language B **express** language A?
  - does $\tau$ **correctly implement** the primitives of A using the primitives of B?

- We focus correctness w.r.t. **contextual equivalence**.
  - equates programs if they **behave the same** (w.r.t. termination) in all contexts
  - it is a generic notion, **applicable for many** programming languages
Goals of the Current Work

\(\pi\)-calculus [Milner, Parrow, & Walker, 1992]
- a standard model for (mobile) processes with message passing
- we use the synchronous \(\pi\)-calculus with replication and a constant \(\text{stop}\) (called \(\Pi_{\text{Stop}}\))

Concurrent Haskell [Peyton-Jones, Gordon, & Finne, 1996]
- extends Haskell by concurrent threads and shared-memory (so-called MVars)
- we use the calculus CH (a variant of CHF, [S. & Schmidt-Schauß, 2011])

Questions:
- Can we encode/translate \(\Pi_{\text{Stop}}\) into CH?
- Which (correctness) properties hold for the translation?
The Source Language $\Pi_{\text{Stop}}$: The $\pi$-calculus with Stop

- we consider the synchronous $\pi$-calculus, with replication, without sums
- extended with a constant $\text{Stop}$ to signal success [S. & Schmidt-Schauß 2015]

**Syntax of Processes**

\[ P, Q \in \text{Proc}_\pi ::= P | Q | x(y).P | \overline{xy}.P | \nu x.P | !P | 0 | \text{Stop} \]

- $x(y).P$: input
- $\overline{xy}.P$: output
- $\nu x.P$: success

Reduction contexts: $D \in \text{PCtxt}_\pi ::= [\cdot] | D | P | P | D | \nu x. D$

Reduction rule for interaction Standard Reduction $sr \rightarrow$: $x(y).P | Q | xz.Q \xrightarrow{ia} P[z/y] | Q \xrightarrow{sr} Q$ if $P \equiv D[P']$, $P' \xrightarrow{ia} Q'$, $D[Q'] \equiv Q$
The Source Language $\Pi_{\text{Stop}}$: The $\pi$-calculus with $\text{Stop}$

- we consider the synchronous $\pi$-calculus, with replication, without sums
- extended with a constant $\text{Stop}$ to signal success [S. & Schmidt-Schauß 2015]

Syntax of Processes

$P, Q \in \text{Proc}_\pi ::= P \mid Q \mid x(y).P \mid \overline{xy}.P \mid \nu x.P \mid !P \mid 0 \mid \text{Stop}$

Reduction contexts: $\mathbb{D} \in PCtxt_\pi ::= [\cdot] \mid \mathbb{D} \mid P \mid P \mid \mathbb{D} \mid \nu x.\mathbb{D}$

Reduction rule for interaction

$x(y).P \mid \overline{x}z.Q \xrightarrow{ia} P[z/y] \mid Q$

Standard Reduction $\xrightarrow{sr}$:

$P \xrightarrow{sr} Q$ if $P \equiv \mathbb{D}[P'], P' \xrightarrow{ia} Q', \mathbb{D}[Q'] \equiv Q$

Process $P$ is successful if $P \equiv \mathbb{D}[\text{Stop}]$
The Target Language CH: Functional Language + Threads & MVars

Syntax of Processes:

\[ P_i \in \text{Proc}_{\text{CH}} ::= P_1 \mid P_2 \mid \nu x. P \mid \leftarrow e \mid x = e \mid x \, m \, e \mid x \, m \rightarrow \]

- thread
- MVars

Main-thread: a unique distinguished thread \( \leftarrow e \)

Syntax of Expressions:

\[ e_i ::= x \mid \lambda x. e \mid (e_1 \, e_2) \mid c \, e_1 \rightarrow \mid \text{case } e \text{ of } \text{alts} \mid \text{seq } e_1 \, e_2 \mid \text{letrec } x_1 = e_1, \ldots, x_n = e_n \text{ in } e \]

- extended lambda-calculus
- IO-monad & concurrency
- monadic MVar-operations

D. Sabel | Synchronous Message-Passing by MVars | EXPRESS/SOS 2020 5/19

Introduction IIStop CH Translations Conclusion
Monadic Computations

(lunit) $\leftarrow M[\text{return } e_1 >> e_2] \rightarrow \leftarrow M[e_2 e_1]

(fork) $\leftarrow M[\text{forkIO } e] \rightarrow \leftarrow M[\text{return }()] | \leftarrow e

(tmvar) $\leftarrow M[\text{takeMVar } x] | x \text{ m e} \rightarrow \leftarrow M[\text{return } e] | x \text{ m }$

(pmvar) $\leftarrow M[\text{putMVar } x e] | x \text{ m } \rightarrow \leftarrow M[\text{return }()] | x \text{ m } e$

... 

Functional Evaluation

(beta) $\leftarrow M[F[\text{((\lambda x.e_1) e_2) }]] \rightarrow \leftarrow M[F[e_1[e_2/x]]]$

... 

Standard Reduction $sr$: 

$P \rightarrow Q$ if $P \equiv D[P'], P' \rightarrow Q', D[Q'] \equiv Q$

Process $P$ is successful if 

$P \equiv \nu x_1 \ldots x_n.( \text{return } e | P')$
Semantics of Processes in Source and Target Language

Observations:

- **P may-converges** \( (P\downarrow) \) iff \( P \xrightarrow{sr,*} P' \) and \( P' \) is successful.

- **P should-converges** \( (P\downarrow) \) iff \( \forall P' : P \xrightarrow{sr,*} P' \implies P' \downarrow \)

Contextual equivalence \( \sim_c \)

\[
P_1 \sim_c P_2 \text{ iff } \forall C : C[P_1]\downarrow \iff C[P_2]\downarrow \text{ and } C[P_1]\downarrow \iff C[P_2]\downarrow
\]
Task: Find a Translation ...

That is correct w.r.t. $\sim_c$

We present the main ideas of the translation step by step:

- Translation of the $\text{Stop}$-constant
- Translation of $0$, parallel composition, replication
- Translation of channels (and interaction): with different variations
Translation of **Stop:**

\[
\tau_0(P) = \text{main} \rightarrow\!
\begin{array}{l}
\text{do } \{ \text{stop }\leftarrow \text{newMVar}(); \\
\text{forkIO } \tau(P); \\
\text{putMVar } \text{stop }(); \} \\
\end{array}
\]

\[
\tau(\text{Stop}) = \text{takeMVar } \text{stop} \\
= C_{out}^\tau[\tau(P)]
\]
Translation of \textbf{Stop:}

\[
\tau_0(P) = \text{main} \downarrow \text{do} \{ \text{stop} \leftarrow \text{newMVar}(); \\
\text{forkIO } \tau(P); \\
\text{putMVar } \text{stop}(); \}
\]

\[
\tau(\text{Stop}) = \text{takeMVar } \text{stop}
\]

\[
= C_{\text{out}}^\tau[\tau(P)]
\]
Translation

**Translation of Stop:**

\[
\tau_0(P) = \text{main} \begin{array}{c}
\text{do} \{ \text{stop} \leftarrow \text{newMVar}(); \text{forkIO } \tau(P); \text{putMVar } \text{stop}() \}\end{array}
\]

\[
\tau(\text{Stop}) = \text{takeMVar } \text{stop}
\]

\[
= C^\tau_{\text{out}}[\tau(P)]
\]
Translation

Translation of Stop:

\[ \tau_0(P) = \texttt{main} \texttt{do} \{ \texttt{stop} \leftarrow \texttt{newMVar}(); \texttt{forkIO} \tau(P); \texttt{putMVar} \texttt{stop}(); \} \]

\[ = C^\tau_{out}[\tau(P)] \]

\[ \tau(\texttt{Stop}) = \texttt{takeMVar \ stop} \]
Translation of $\text{Stop}$:

$$\tau_0(P) = \text{main} \begin{array}{l} \text{do} \{ \text{stop} \leftarrow \text{newMVar}(); \} \\
\text{forkI0 } \tau(P); \\
\text{putMVar } \text{stop}(); \} \\
= C^\tau_{\text{out}}[\tau(P)]$$

$$\tau(\text{Stop}) = \text{takeMVar } \text{stop}$$
Translation

Translation of Stop:

\[
\tau_0(P) = \begin{array}{c}
\text{main} \\
\text{do}
\end{array} \\
\{ \text{stop} \leftarrow \text{newMVar}(); \\
\text{forkIO } \tau(P); \\
\text{putMVar } \text{stop}(); \}
\]

\[
\tau(\text{Stop}) = \text{takeMVar } \text{stop}
\]

\[
= C^\tau_{\text{out}}[\tau(P)]
\]
Translation of \textbf{Stop:}

\[
\tau_0(P) = \langle \text{main} \rangle \text{ do } \{ \text{stop} \leftarrow \text{newMVar}(); \text{forkIO } \tau(P); \text{putMVar } \text{stop}(); \}\n\]

\[
\tau(\text{Stop}) = \text{takeMVar } \text{stop}
\]

\[
= C^\tau_{out}[\tau(P)]
\]
Translation

**Translation of Stop:**

\[ \tau_0(P) = \left\{ \begin{array}{l}
\text{main} \\
\text{do}
\end{array} \right. \}
\]

\[ \left\{ \begin{array}{l}
\text{stop} \leftarrow \text{newMVar} () \\
\text{forkIO } \tau(P) \\
\text{putMVar } stop ()
\end{array} \right. \]

\[ = C^r_{out}[\tau(P)] \]

\[ \tau(\text{Stop}) = \text{takeMVar } stop \]

---

![Diagram showing the flow of control and data between main and other threads.]

- Main thread
- Other threads
- Stop
- put

D. Sabel | Synchronous Message-Passing by MVars | EXPRESS/SOS 2020 | 9/19
Translation

Translation of \textbf{Stop}:

\[
\tau_0(P) = \begin{array}{ccc}
\text{main} & \text{do} & \{\text{stop} \leftarrow \text{newMVar} (); \\
& \text{forkIO } \tau(P); \\
& \text{putMVar } \text{stop} ()\}
\end{array} \quad \tau(\text{Stop}) = \text{takeMVar } \text{stop}
\]

\[= C_{\text{out}}^{\tau}[\tau(P)]\]
Translation

**Translation of Stop:**

\[
\tau_0(P) = \text{main} \begin{array}{l}
do \{ \text{stop} \leftarrow \text{newMVar}(); \\
\text{forkIO } \tau(P); \\
\text{putMVar } \text{stop}(); \\
\} \\
\end{array}
\]

\[= C^\tau_{out} [\tau(P)] \]

**Translation of 0, Parallel Composition, and Replication:**

\[
\tau(0) = \text{return}() \\
\tau(P \parallel Q) = \text{do } \{ \text{forkIO } \tau(Q); \tau(P) \} \\
\tau(!P) = \text{letrec } f = \text{do } \{ \text{forkIO } \tau(P); f \} \text{ in } f
\]
Translation of Channels and Message Passing

Two approaches to encode synchronous communication by several accesses to MVars

- Using **a private MVar** per communication
  
  (similar to [Boudol 1992, Honda & Tokora, 1991] where private names guarantee correct communication while encoding the synchronous in the asynchronous $\pi$-calculus)

- Using a fixed number of **global MVars** per channel
  
  avoids to dynamically generate “garbage”
Translation with Private MVar

\(\pi\)-calculus-channels are translated into

\[
\text{data Channel} \equiv \text{Chan} \ (\text{MVar} \ (\text{Channel}, \text{MVar} \ ()))
\]

\[
\tau(\nu x.P) = \text{do} \{ \text{chan}x \leftarrow \text{newEmptyMVar}; \text{letrec} \ x = \text{Chan chan}x \ \text{in} \ \tau(P) \}
\]

\[
\tau(\overline{x} z.Q) = \text{do} \{ \text{check}x \leftarrow \text{newMVar} ();
\text{putMVar} \ (\overline{\text{unchan}} \ x) \ (z, \text{check}x);
\text{putMVar} \ \text{check}x (); \tau(Q) \}
\]

\[
\tau(x(y).P) = \text{do} \{ (y, \text{check}x) \leftarrow \text{takeMVar} \ (\overline{\text{unchan}} \ x); \text{takeMVar} \ \text{check}x; \tau(P) \}
\]
Translation with Private MVar

π-calculus-channels are translated into

```
data Channel = Chan (MVar (Channel, MVar ()))
```

```
τ(νx.P) = do {chanx ← newEmptyMVar; letrec x = Chan chanx in τ(P)}
```

```
τ(νx.P) = do {
  checkx ← newMVar ();
  putMVar (unchan x) (z, checkx);
  putMVar checkx (); τ(Q)}
```

```
τ(νx.P) = do {
  (y, checkx) ← takeMVar (unchan x);
  putMVar checkx (); τ(P)}
```

```
τ(y.P) = do {
  (y, checkx) ← takeMVar (unchan x);
  τ(Q)}
```

```
τ(νx.P) = do {
  chanx ← newEmptyMVar;
  letrec x = Chan chanx in τ(P)}
```

```
τ(νx.P) = do {
  checkx ← newMVar ();
  putMVar (unchan x) (z, checkx);
  putMVar checkx (); τ(Q)}
```

```
τ(νx.P) = do {
  (y, checkx) ← takeMVar (unchan x);
  putMVar checkx (); τ(P)}
```

```
τ(y.P) = do {
  (y, checkx) ← takeMVar (unchan x);
  putMVar checkx (); τ(P)}
```

```
τ(νx.P) = do {
  chanx ← newEmptyMVar;
  letrec x = Chan chanx in τ(P)}
```

```
τ(νx.P) = do {
  checkx ← newMVar ();
  putMVar (unchan x) (z, checkx);
  putMVar checkx (); τ(Q)}
```

```
τ(νx.P) = do {
  (y, checkx) ← takeMVar (unchan x);
  putMVar checkx (); τ(P)}
```

```
τ(y.P) = do {
  (y, checkx) ← takeMVar (unchan x);
  putMVar checkx (); τ(P)}
```

```
τ(νx.P) = do {
  chanx ← newEmptyMVar;
  letrec x = Chan chanx in τ(P)}
```

```
τ(νx.P) = do {
  checkx ← newMVar ();
  putMVar (unchan x) (z, checkx);
  putMVar checkx (); τ(Q)}
```

```
τ(νx.P) = do {
  (y, checkx) ← takeMVar (unchan x);
  putMVar checkx (); τ(P)}
```

```
τ(y.P) = do {
  (y, checkx) ← takeMVar (unchan x);
  putMVar checkx (); τ(P)}
```

```
τ(νx.P) = do {
  chanx ← newEmptyMVar;
  letrec x = Chan chanx in τ(P)}
```

```
τ(νx.P) = do {
  checkx ← newMVar ();
  putMVar (unchan x) (z, checkx);
  putMVar checkx (); τ(Q)}
```

```
τ(νx.P) = do {
  (y, checkx) ← takeMVar (unchan x);
  putMVar checkx (); τ(P)}
```

```
τ(y.P) = do {
  (y, checkx) ← takeMVar (unchan x);
  putMVar checkx (); τ(P)}
```

```
τ(νx.P) = do {
  chanx ← newEmptyMVar;
  letrec x = Chan chanx in τ(P)}
```

```
τ(νx.P) = do {
  checkx ← newMVar ();
  putMVar (unchan x) (z, checkx);
  putMVar checkx (); τ(Q)}
```

```
τ(νx.P) = do {
  (y, checkx) ← takeMVar (unchan x);
  putMVar checkx (); τ(P)}
```

```
τ(y.P) = do {
  (y, checkx) ← takeMVar (unchan x);
  putMVar checkx (); τ(P)}
```

```
τ(νx.P) = do {
  chanx ← newEmptyMVar;
  letrec x = Chan chanx in τ(P)}
```

```
τ(νx.P) = do {
  checkx ← newMVar ();
  putMVar (unchan x) (z, checkx);
  putMVar checkx (); τ(Q)}
```

```
τ(νx.P) = do {
  (y, checkx) ← takeMVar (unchan x);
  putMVar checkx (); τ(P)}
```

```
τ(y.P) = do {
  (y, checkx) ← takeMVar (unchan x);
  putMVar checkx (); τ(P)}
```

```
τ(νx.P) = do {
  chanx ← newEmptyMVar;
  letrec x = Chan chanx in τ(P)}
```

```
τ(νx.P) = do {
  checkx ← newMVar ();
  putMVar (unchan x) (z, checkx);
  putMVar checkx (); τ(Q)}
```

```
τ(νx.P) = do {
  (y, checkx) ← takeMVar (unchan x);
  putMVar checkx (); τ(P)}
```

```
τ(y.P) = do {
  (y, checkx) ← takeMVar (unchan x);
  putMVar checkx (); τ(P)}
```

```
τ(νx.P) = do {
  chanx ← newEmptyMVar;
  letrec x = Chan chanx in τ(P)}
```

```
τ(νx.P) = do {
  checkx ← newMVar ();
  putMVar (unchan x) (z, checkx);
  putMVar checkx (); τ(Q)}
```

```
τ(νx.P) = do {
  (y, checkx) ← takeMVar (unchan x);
  putMVar checkx (); τ(P)}
```

```
τ(y.P) = do {
  (y, checkx) ← takeMVar (unchan x);
  putMVar checkx (); τ(P)}
```

```
τ(νx.P) = do {
  chanx ← newEmptyMVar;
  letrec x = Chan chanx in τ(P)}
```

```
τ(νx.P) = do {
  checkx ← newMVar ();
  putMVar (unchan x) (z, checkx);
  putMVar checkx (); τ(Q)}
```

```
τ(νx.P) = do {
  (y, checkx) ← takeMVar (unchan x);
  putMVar checkx (); τ(P)}
```

```
τ(y.P) = do {
  (y, checkx) ← takeMVar (unchan x);
  putMVar checkx (); τ(P)}
```

```
τ(νx.P) = do {
  chanx ← newEmptyMVar;
  letrec x = Chan chanx in τ(P)}
```

```
τ(νx.P) = do {
  checkx ← newMVar ();
  putMVar (unchan x) (z, checkx);
  putMVar checkx (); τ(Q)}
```

```
τ(νx.P) = do {
  (y, checkx) ← takeMVar (unchan x);
  putMVar checkx (); τ(P)}
```

```
τ(y.P) = do {
  (y, checkx) ← takeMVar (unchan x);
  putMVar checkx (); τ(P)}
```

```
τ(νx.P) = do {
  chanx ← newEmptyMVar;
  letrec x = Chan chanx in τ(P)}
```

```
τ(νx.P) = do {
  checkx ← newMVar ();
  putMVar (unchan x) (z, checkx);
  putMVar checkx (); τ(Q)}
```

```
τ(νx.P) = do {
  (y, checkx) ← takeMVar (unchan x);
  putMVar checkx (); τ(P)}
```

```
τ(y.P) = do {
  (y, checkx) ← takeMVar (unchan x);
  putMVar checkx (); τ(P)}
```

```
τ(νx.P) = do {
  chanx ← newEmptyMVar;
  letrec x = Chan chanx in τ(P)}
```

```
τ(νx.P) = do {
  checkx ← newMVar ();
  putMVar (unchan x) (z, checkx);
  putMVar checkx (); τ(Q)}
```

```
τ(νx.P) = do {
  (y, checkx) ← takeMVar (unchan x);
  putMVar checkx (); τ(P)}
```

```
τ(y.P) = do {
  (y, checkx) ← takeMVar (unchan x);
  putMVar checkx (); τ(P)}
```

```
τ(νx.P) = do {
  chanx ← newEmptyMVar;
  letrec x = Chan chanx in τ(P)}
```

```
τ(νx.P) = do {
  checkx ← newMVar ();
  putMVar (unchan x) (z, checkx);
  putMVar checkx (); τ(Q)}
```

```
τ(νx.P) = do {
  (y, checkx) ← takeMVar (unchan x);
  putMVar checkx (); τ(P)}
```

```
τ(y.P) = do {
  (y, checkx) ← takeMVar (unchan x);
  putMVar checkx (); τ(P)}
```
Translation with Private MVar

\(\pi\)-calculus-channels are translated into

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

\[
\tau(\nu x.P) = \text{ do } \{ \text{ chanx } \leftarrow \text{ newEmptyMVar}; \text{ letrec } x = \text{ Chan chanx in } \tau(P) \}
\]

\[
\tau(\bar{x}z.Q) = \text{ do } \{ \text{ checkx } \leftarrow \text{ newMVar } (); \text{ putMVar } (\text{ unchan } x) (z, \text{ checkx }); \text{ putMVar } \text{ checkx } (); \tau(Q) \}
\]

\[
\tau(x(y).P) = \text{ do } \{ (y, \text{ checkx }) \leftarrow \text{ takeMVar } (\text{ unchan } x); \text{ takeMVar } \text{ checkx }; \tau(P) \}
\]

sender
\(\bar{x}z.Q\)

receiver
\(x(y).P\)
Translation with Private MVar

$\pi$-calculus-channels are translated into

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

\[
\tau(\nu x.P) = \text{ do } \{ \text{ chanx } \leftarrow \text{ newEmptyMVar}; \text{ letrec } x = \text{ Chan} \text{ chanx } \text{ in } \tau(P) \}
\]

\[
\tau(\overline{x}z.Q) = \text{ do } \{ \text{ checkx } \leftarrow \text{ newMVar } (); \quad \tau(x(y).P) = \text{ do } \{(y, \text{ checkx}) \leftarrow \text{ takeMVar } (\text{ unchan} x); \quad \text{ takeMVar} \text{ checkx}; \tau(P)\}
\]

\[
\begin{array}{c}
sender \\
\overline{x}z.Q \\
\end{array}
\]

\[
\begin{array}{c}
x \\
\text{ take} \\
\frac{\text{ blocked}}{\text{ receiver}} \\
x(y).P
\end{array}
\]
Translation with Private MVar

\(\pi\)-calculus-channels are translated into

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

\[
\tau(\nu x. P) = \text{do}\{\text{chanx} \leftarrow \text{newEmptyMVar} ; \text{letrec} \ x = \text{Chan} \ chanx \ \text{in} \ \tau(P)\}
\]

\[
\tau(\overline{x}z. Q) = \text{do}\{\text{checkx} \leftarrow \text{newMVar} () ; \\
\quad \text{putMVar} (\text{unchan} x) (z, \text{checkx}) ; \\
\quad \text{putMVar} \ \text{checkx} () ; \tau(Q)\}
\]

\[
\tau(x(y). P) = \text{do}\{(y, \text{checkx}) \leftarrow \text{takeMVar} (\text{unchan} x) ; \\
\quad \text{takeMVar} \ \text{checkx} ; \tau(P)\}
\]
Translation with Private MVar

\(\pi\)-calculus-channels are translated into

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

\[
\tau(\nu x. P) = \text{do} \{ \text{chanx} \leftarrow \text{newEmptyMVar}; \text{letrec } x = \text{Chan chanx in } \tau(P) \}
\]

\[
\tau(\overline{x} z. Q) = \text{do} \{ \text{checkx} \leftarrow \text{newMVar} (); \begin{align*}
\text{putMVar (unchan x) (z, checkx)}; \\
\text{putMVar checkx (); } \tau(Q)
\end{align*}
\]

\[
\tau(x(y). P) = \text{do} \{ (y, checkx) \leftarrow \text{takeMVar (unchan x); takeMVar checkx; } \tau(P) \}
\]
Translation with Private MVar

$\pi$-calculus-channels are translated into

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

$\tau(\nu x. P) = \text{do} \{ \text{chanx} \leftarrow \text{newEmptyMVar}; \text{letrec} \ x = \text{Chan chanx} \ \text{in} \ \tau(P) \}$

$\tau(\overline{xz}. Q) = \text{do} \{ \text{checkx} \leftarrow \text{newMVar}(); \text{putMVar} (\text{unchan x}) (z, \text{checkx}); \text{putMVar} \ \text{checkx} (); \tau(Q) \}$

$\tau(x(y). P) = \text{do} \{(y, \text{checkx}) \leftarrow \text{takeMVar} (\text{unchan x}); \text{takeMVar} \ \text{checkx}; \tau(P) \}$

\[
\text{sender} \\
\overline{xz}. Q
\]

\[
\text{receiver} \\
x(y). P
\]

\[
\text{checkx}
\]

\[
(x) \text{ take} \\
(z, \_)
\]
Translation with Private MVar

\(\pi\)-calculus-channels are translated into

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

\[
\tau(\nu x. P) = \text{ do } \{ \text{ chanx } \leftarrow \text{ newEmptyMVar } ; \text{ letrec } x = \text{ Chan } \text{ chanx } \text{ in } \tau(P) \}
\]

\[
\tau(\bar{x}z.Q) = \text{ do } \{ \text{ checkx } \leftarrow \text{ newMVar } () ; \text{ putMVar } (\text{ unchan x}) (z, \text{ checkx}) ; \text{ putMVar } \text{ checkx } () ; \tau(Q) \}
\]

\[
\tau(x(y). P) = \text{ do } \{ (y, \text{ checkx}) \leftarrow \text{ takeMVar } (\text{ unchan x}) ; \text{ takeMVar } \text{ checkx } ; \tau(P) \}
\]
Translation with Private MVar

$\pi$-calculus-channels are translated into

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

\[
\tau(\nu x. P) = \text{do } \{ \text{chanx } \leftarrow \text{newEmptyMVar}; \text{letrec } x = \text{Chan chanx in } \tau(P) \}
\]

\[
\tau(\overline{x} z. Q) = \text{do } \{ \text{checkx } \leftarrow \text{newMVar ()}; \\
\quad \text{putMVar (unchan x) (z, checkx)}; \\
\quad \text{putMVar checkx (); } \tau(Q) \}
\]

\[
\tau(x(y). P) = \text{do } \{ (y, \text{checkx}) \leftarrow \text{takeMVar (unchan x)}; \\
\quad \text{takeMVar checkx; } \tau(P) \}
\]
Translation with Private MVar

\(\pi\)-calculus-channels are translated into

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

\[
\tau(\nu x.P) = \text{do} \{\text{chan} x \leftarrow \text{newEmptyMVar}; \ \text{letrec} \ x = \text{Chan} \ \text{chan} x \ \text{in} \ \tau(P)\}
\]

\[
\tau(\overline{x}z.Q) = \text{do} \{\text{check} x \leftarrow \text{newMVar} (); \ \tau(x(y).P) = \text{do} \{(y, \text{check} x) \leftarrow \text{takeMVar} \ (\text{unchan} x); \ \text{takeMVar} \ \text{check} x; \ \tau(P)\}
\]

\[
\tau(x(y).P) = \text{do} \{(y, \text{check} x) \leftarrow \text{takeMVar} \ (\text{unchan} x); \ \text{takeMVar} \ \text{check} x; \ \tau(P)\}
\]

\[
\tau(x(y).P) = \text{do} \{(y, \text{check} x) \leftarrow \text{takeMVar} \ (\text{unchan} x); \ \text{takeMVar} \ \text{check} x; \ \tau(P)\}
\]

sender

\(\overline{x}z.Q\)

\(\llbracket\text{blocked}\rrbracket\)

put

\(\llbracket\text{check} x\rrbracket\)

take

receiver

\(x(y).P\)
Translation with Private MVar

\(\pi\)-calculus-channels are translated into

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

\[
\tau(\nu x.P) = \text{do} \{ \text{chanx} \leftarrow \text{newEmptyMVar}; \text{letrec} \ x = \text{Chan} \ \text{chanx} \ \text{in} \ \tau(P) \}
\]

\[
\tau(\overline{x} z.Q) = \text{do} \{ \text{checkx} \leftarrow \text{newMVar} (); \ 
\text{putMVar} \ (\text{unchan} \ x) \ (z, \text{checkx}); \ 
\text{putMVar} \ \text{checkx} (); \ tau(\text{Q}) \}
\]

\[
\tau(x(y).P) = \text{do} \{ (y, \text{checkx}) \leftarrow \text{takeMVar} \ (\text{unchan} \ x); \ 
\text{takeMVar} \ \text{checkx}; \ tau(\text{P}) \}
\]
Translation with Private MVar

\(\pi\)-calculus-channels are translated into

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

\[
\tau(\nu x. P) = \text{ do } \{ \text{ chanx } \leftarrow \text{ newEmptyMVar}; \ \text{ letrec } \ x = \text{ Chan chanx } \ \text{ in } \ \tau(P) \}
\]

\[
\tau(\overline{x}z. Q) = \text{ do } \{ \text{ checkx } \leftarrow \text{ newMVar } (); \text{ putMVar} \ (\text{ unchan x }) \ (z, \text{ checkx}); \text{ putMVar} \ \text{ checkx } (); \ \tau(Q) \}
\]

\[
\tau(x(y). P) = \text{ do } \{ (y, \text{ checkx}) \leftarrow \text{ takeMVar} \ (\text{ unchan x}); \text{ takeMVar} \ \text{ checkx} ; \ \tau(P) \}
\]

sender

\(\overline{x}z. Q\)

receiver

\(x(y). P\)

\[x\]

\[\text{ checkx}\]
$$\tau_0(P) = C^\tau_{out}[\tau(p)]$$

$$\tau(\text{Stop}) = \text{takeMVar \ stop}$$

$$\tau(0) = \text{return}()$$

$$\tau(P \mid Q) = \text{do \{forkIO \ \tau(Q); \tau(P)\}}$$

$$\tau(!P) = \text{letrec \ } f = \text{do \{forkIO \ \tau(P); \theta \}} \ \text{in} \ f$$

$$\tau(\nu x. P) = \text{do \{chanx \leftarrow newEmptyMVar; letrec \ } x = \text{Chan chanx \ in} \ \tau(P)\}$$

$$\tau(\bar{x}z.Q) = \text{do \{checkx \leftarrow newMVar();}$$

$$\text{putMVar (unchan x) (z, checkx);}$$

$$\text{\{putMVar checkx (); \tau(Q)\}}$$

$$\tau(x(y).P) = \text{do \{(y, checkx) \leftarrow takeMVar (unchan x);}$$

$$\text{takeMVar checkx; \tau(P)\}}$$
Correctness of Translation $\tau$

**Theorem (Convergence Equivalence)**

For closed $P \in \Pi_{\text{Stop}}$: $P \downarrow \iff C_{\text{out}}^\tau[\tau(P)] \downarrow$ and $P \downarrow \iff C_{\text{out}}^\tau[\tau(P)] \downarrow$

Proof consists of four parts:

- ("$\downarrow \Rightarrow \downarrow$") $P \xrightarrow{\text{sr},*} P', P'$ successful $\implies \exists Q : C_{\text{out}}^\tau[\tau(P)] \xrightarrow{\text{sr},*} Q, Q$ successful.

- ("$\downarrow \Leftarrow \downarrow$") $C_{\text{out}}^\tau[\tau(P)] \xrightarrow{\text{sr},*} Q, Q$ successful $\implies \exists P' : P \xrightarrow{\text{sr},*} P', P'$ successful.

- ("$\downarrow \Leftarrow \uparrow$") $P \xrightarrow{\text{sr},*} P', P' \uparrow$ $\implies \exists Q : C_{\text{out}}^\tau[\tau(P)] \xrightarrow{\text{sr},*} Q, Q \uparrow$.

- ("$\downarrow \Rightarrow \uparrow$") $C_{\text{out}}^\tau[\tau(P)] \xrightarrow{\text{sr},*} Q, Q \uparrow$ $\implies \exists P' : P \xrightarrow{\text{sr},*} P', P' \uparrow$

All parts require to inductively construct reduction sequences from given ones.

For parts ("$\downarrow \Leftarrow \downarrow$") and ("$\downarrow \Rightarrow \downarrow$"), the given sequences $C_{\text{out}}^\tau[\tau(P)] \xrightarrow{\text{sr},*} Q$ have to be reordered, cut and/or extended to “back-translate” them.
Theorem (Adequacy)

Translation $\tau$ is adequate, i.e. for all $P, P' \in \Pi_{\text{Stop}}$: $\tau(P) \sim_{c, \tau_0} \tau(P') \implies P \sim_c P'$

Theorem

The translation $\tau$ is not fully abstract ($P \sim_c P' \iff \tau(P) \sim_{c, \tau_0} \tau(P')$).

On closed processes $P, P'$: $P \sim_c P' \iff \tau(P) \sim_{c, \tau_0} \tau(P')$

where $e_1 \sim_{c, \tau_0} e_2$ iff for all $C : FV(C[e_1]) \cup FV(C[e_2]) \subseteq \{\text{stop}\}$:

$C^\tau_{\text{out}}[C[e_1]] \downarrow \iff C^\tau_{\text{out}}[C[e_2]] \downarrow$ and $C^\tau_{\text{out}}[C[e_1]] \downarrow \iff C^\tau_{\text{out}}[C[e_2]] \downarrow$
Translations with Global MVars

Ideas:
- Translation of stop, 0, 1, ! as before
- $\pi$-calculus-channels are translated into data of type

\[
data \ \text{Channel} = \text{Chan} \left( \text{MVar Channel} \right) \left( \text{MVar } () \right) \ldots \left( \text{MVar } () \right)
\]

i.e. channel $x$ becomes a binding $x = \text{Chan content check}_1 \ldots \text{check}_n$

- MVars content, check$_1$, ..., check$_n$ are created once and are (globally) visible via $x$
- Programs for sender $xz$ and receiver $x(y)$ are restricted:
  - They exchange the message via the content-MVar
  - They perform takeMVar & putMVar on the check-MVars for synchronisation
Translations with Global MVars (Cont’d)

Reminder: a channel $x$ becomes a binding $x = \text{Chan content check}_1 \ldots \text{check}_n$

Questions:
- Are there correct translations under these restrictions?
- How many check-MVars are required?
- What is the smallest correct translation?

Approach:
- enumerate all translations and automatically search for counter-examples
- check correctness of the remaining (potentially correct) translations by hand

Conjecture (Proved in the meantime, not yet published)
With the described restrictions two check-MVars are required.
Correct Translation with Two Check-MVars

\[ T_1(\overline{x}z.Q) = \text{do } \{ \text{putMVar} (\text{check}_1 x) (), \]
\[ \text{putMVar} (\text{content } x) z; \]
\[ \text{takeMVar} (\text{check}_2 x); \]
\[ \text{takeMVar} (\text{check}_1 x); T_1(Q) \} \]

\[ T_1(x(y).P) = \text{do } \{ y \leftarrow \text{takeMVar} (\text{content } x); \]
\[ \text{putMVar} (\text{check}_2 x); T_1(P) \} \]
Correct Translation with Two Check-MVars

\[ T_1(\bar{x}z.Q) = \text{do} \{ \text{putMVar}(\text{check}_1 x)(), \]
\[ \text{putMVar}(\text{content } x)z; \]
\[ \text{takeMVar}(\text{check}_2 x); \]
\[ \text{takeMVar}(\text{check}_1 x); T_1(Q) \} \]

\[ T_1(x(y).P) = \text{do} \{ y \leftarrow \text{takeMVar}(\text{content } x); \]
\[ \text{putMVar}(\text{check}_2 x); T_1(P) \} \]
Correct Translation with Two Check-MVars

\[ T_1(\overline{x}z.Q) = \textbf{do} \{ \text{putMVar} (check_1 x) (), \]
\[ \quad \text{putMVar} (content x) z; \]
\[ \quad \text{takeMVar} (check_2 x); \]
\[ \quad \text{takeMVar} (check_1 x); T_1(Q) \} \]

\[ T_1(x(y).P) = \textbf{do} \{ y \leftarrow \text{takeMVar} (content x); \]
\[ \quad \text{putMVar} (check_2 x); T_1(P) \} \]
Correct Translation with Two Check-MVars

\[ T_1(\overline{x}z.Q) = \begin{array}{l}
\text{do} \{ \text{putMVar}(\text{check}_1 x) (); \\
\text{putMVar}(\text{content } x) z; \\
\text{takeMVar}(\text{check}_2 x); \\
\text{takeMVar}(\text{check}_1 x); T_1(Q) \} 
\end{array} \]

\[ T_1(x(y).P) = \begin{array}{l}
\text{do} \{ y \leftarrow \text{takeMVar}(\text{content } x); \\
\text{putMVar}(\text{check}_2 x); T_1(P) \} 
\end{array} \]
Correct Translation with Two Check-MVars

\[ T_1(xz.Q) = \text{do} \{ \text{putMVar}(\text{check}_1 x)(); \]
\[ \text{putMVar}((\text{content} x) z; \]
\[ \text{takeMVar}(\text{check}_2 x); \]
\[ \text{takeMVar}(\text{check}_1 x); T_1(Q) \} \]

\[ T_1(x(y).P) = \text{do} \{ y \leftarrow \text{takeMVar}(\text{content} x); \]
\[ \text{putMVar}(\text{check}_2 x); T_1(P) \} \]
Correct Translation with Two Check-MVars

\[ T_1(xz.Q) = \text{do} \{ \text{putMVar (check}_1 x)(()), \text{putMVar (content } x) z; \text{takeMVar (check}_2 x); \text{takeMVar (check}_1 x); T_1(Q) \} \]

\[ T_1(x(y).P) = \text{do} \{ y \leftarrow \text{takeMVar (content } x); \text{putMVar (check}_2 x); T_1(P) \} \]
Correct Translation with Two Check-MVars

\[ T_1(\overline{xyz}.Q) = \text{do} \{\text{putMVar (check}_1 x)(), \text{putMVar (content x) z; takeMVar (check}_2 x); \text{takeMVar (check}_1 x); T_1(Q)\} \]

\[ T_1(x(y).P) = \text{do} \{y \leftarrow \text{takeMVar (content x); putMVar (check}_2 x); T_1(P)\} \]
Correct Translation with Two Check-MVars

\[ T_1(\overline{xyz}.Q) = \text{do} \{ \text{putMVar}(\text{check}_1 x)(); \\
\text{putMVar}(\text{content } x) z; \\
\text{takeMVar}(\text{check}_2 x); \\
\text{takeMVar}(\text{check}_1 x); T_1(Q) \} \]

\[ T_1(x(y).P) = \text{do} \{ y \leftarrow \text{takeMVar}(\text{content } x); \\
\text{putMVar}(\text{check}_2 x); T_1(P) \} \]
Correct Translation with Two Check-MVars

\[ T_1(\overline{xz}.Q) = \text{do}\left\{ \text{putMVar}(\text{check}_1 x)(), \text{putMVar}(\text{content}\ x)\ z; \ 
\text{takeMVar}(\text{check}_2 x); \ 
\text{takeMVar}(\text{check}_1 x); T_1(Q) \right\} \]

\[ T_1(x(y).P) = \text{do}\left\{ y \leftarrow \text{takeMVar}(\text{content}\ x); \ 
\text{putMVar}(\text{check}_2 x); T_1(P) \right\} \]
Correct Translation with Two Check-MVars

\[ T_1(\overline{x}z. Q) = \text{do} \{ \text{putMVar}(\text{check}_1 x)(); \]
\[
\text{putMVar}(\text{content } x) z; \]
\[
\text{takeMVar}(\text{check}_2 x); \]
\[
\text{takeMVar}(\text{check}_1 x); T_1(Q) \} \]

\[ T_1(x(y). P) = \text{do} \{ y \leftarrow \text{takeMVar}(\text{content } x); \]
\[
\text{putMVar}(\text{check}_2 x); T_1(P) \} \]
**Correct Translation with Two Check-MVars**

\[
T_1(\overline{x}z.Q) = \text{do } \{ \text{putMVar}(\text{check}_1 x)(), \\
\text{putMVar}(\text{content } x) z; \\
\text{takeMVar}(\text{check}_2 x); \\
\text{takeMVar}(\text{check}_1 x); T_1(Q) \}
\]

\[
T_1(x(y).P) = \text{do } \{ y \leftarrow \text{takeMVar}(\text{content } x); \\
\text{putMVar}(\text{check}_2 x); T_1(P) \}
\]
**Correct Translation with Two Check-MVars**

\[ T_1(\bar{x}z.Q) = \text{do} \{ \text{putMVar (check}_1 x)(), \]
\[ \quad \text{putMVar (content} x) z; \]
\[ \quad \text{takeMVar (check}_2 x); \]
\[ \quad \text{takeMVar (check}_1 x); T_1(Q) \} \]

\[ T_1(x(y).P) = \text{do} \{ y \leftarrow \text{takeMVar (content} x); \]
\[ \quad \text{putMVar (check}_2 x); T_1(P) \} \]

---

**Graphical Representation**

```
  content x
  ↖            ↖
  |            |      check2 x
sender  ↖          ↖      receiver
  bar x . Q   check1 x
              ↖            ↖
              |            |      x (y) . P
  ```
Correct Translation with Two Check-MVars

\[ T_1(\bar{x}z.Q) = \texttt{do}\ \{ \texttt{putMVar}(\texttt{check}_1 \ x)(); \}
\texttt{putMVar}(\texttt{content} \ x) \ z; \\
\texttt{takeMVar}(\texttt{check}_2 \ x); \\
\texttt{takeMVar}(\texttt{check}_1 \ x); T_1(Q) \} \]

\[ T_1(x(y).P) = \texttt{do}\ \{ y \leftarrow \texttt{takeMVar}(\texttt{content} \ x); \\
\texttt{putMVar}(\texttt{check}_2 \ x); T_1(P) \} \]

**Theorem**

\( T_1 \) is convergence-equivalent, adequate, and on closed processes also fully-abstract.

Main arguments:

- \( \texttt{MVar}(\texttt{check}_1 \ x) \) is used as a mutex for the receivers on \( x \)
- execution of the sender/receiver protocol is non-overlapping
Interprocess restriction:
One put/take-pair for each check-MVar and it is distributed between sender/receiver.
Translations with Global MVars and Interprocess Restriction

Interprocess restriction:
One put/take-pair for each check-MVar and it is distributed between sender/receiver.

Theorem
Under the interprocess restriction, three check-MVars are necessary and sufficient.

Correct translation:
\[
T_2(\bar{x}z.Q) = \begin{array}{l}
\text{do} \\
\{ \text{putMVar}(\text{content } x) \ z; \\
\text{putMVar}(\text{check}_1 x) (); \\
\text{takeMVar}(\text{check}_2 x); \\
\text{putMVar}(\text{check}_3 x) (); T_2(Q) \} \\
T_2(x(y)P) = \begin{array}{l}
\text{do} \\
\{ \text{takeMVar}(\text{check}_1 x); \\
\text{putMVar}(\text{check}_2 x); \\
\text{takeMVar}(\text{check}_3 x); \\
y \leftarrow \text{takeMVar}(\text{content } x); T_2(P) \}
\end{array}
\]

Results of the automated search for counter-examples:
- for 1 check-MVar 8 of 8 translations are refuted
- for 2 check-MVAr 72 of 72 translations are refuted
- for 3 check-MVAr 762 of 768 translations are refuted
Conclusion & Future Work

Conclusion

- Correct translations from $\Pi_{\text{Stop}}$ into Concurrent Haskell
- Translation with private MVars
- Smallest translations with global MVars
- Translations are convergence equivalent and adequate (fully abstract on closed processes)
- Refuted incorrect translations by automated search for counter-examples

Future Work

- Variations and extensions of $\Pi_{\text{Stop}}$ (recursion, sums, name matching, . . .)
- Other target languages?
- Publish proof of conjecture