Concurrent Programming Languages and Semantic Analyses

Manfred Schmidt-Schauss

Goethe-Universität Frankfurt, Institut für Informatik, Germany

RTA/TLCA 14

Based on joint work with David Sabel
Main Parts

- Diagrams and correctness of transformations
- Concurrency, non-determinism and contextual semantics
- Correctness of a concurrent implementation
Main Parts

- Introduction
- Diagrams and correctness of transformations
  contextual equivalence, diagrams, correctness proofs, meta-rewriting sequences, automation
  LR (a deterministic calculus)
- Concurrency, non-determinism and contextual semantics
  may and should convergence and contextual equivalences, conservativity
  CHF (a concurrent calculus)
- Correctness of a concurrent implementation
  a complex real-world calculus: showing correctness using operational methods
  CSHF (concurrent implementation of software transactional memory)
Alternative semantics approaches, also under concurrency

- denotational semantics
- translations into pi-calculus or other models
- simulation / bisimulation
- logical approaches
- observational semantics / contextual semantics
Question?

- Is there a best / standard semantics?
Question?

- Is there a best / standard semantics?
  - Yes for deterministic programming languages
Question?

- Is there a best / standard semantics?
  
  - Yes for deterministic programming languages
  
  - No for non-deterministic and/or concurrent programming languages
**Question?**

- Is there a best / standard semantics?

- Yes  for deterministic programming languages

- No   for non-deterministic and/or concurrent programming languages

- But  there are good choices
\( e_1 \leq e_2 \iff \forall C : C[e_1] \downarrow \implies C[e_2] \downarrow \)
\( e_1 \sim e_2 \iff e_1 \leq e_2 \text{ and } e_2 \leq e_1 \)

Where:
- \( e_i \) expressions resp. programs
- \( C \) contexts: programs with a hole
- \( e \downarrow \) \( e \) reduces to a successful program

reduction: a fixed-strategy-rewriting of programs.

\( \leq \) contextual approximation
\( \sim \) contextual equivalence

Morris’ contextual equivalence (thesis, 1968)
\[ e_1 \leq_{\text{may}} e_2 \iff \forall C : C[e_1] \downarrow_{\text{may}} \implies C[e_2] \downarrow_{\text{may}} \]
\[ e_1 \sim_{\text{may}} e_2 \iff e_1 \leq_{\text{may}} e_2 \text{ and } e_2 \leq_{\text{may}} e_1 \]

Where:
- \( e_i \): expressions resp. programs
- \( C \): contexts: programs with a hole
- \( e \downarrow_{\text{may}} \): \( e \) may reduce to a successful program (may-convergence)
- reduction: a fixed-strategy-rewriting of programs.

\( \leq_{\text{may}} \): contextual approximation
\( \sim_{\text{may}} \): contextual equivalence

Morris’ contextual equivalence (thesis, 1968)
Q1: True $\not\sim$ False?
**Q1:** True \(\not\sim\) False ?

One context suffices:

\[ C[.] = \text{if } [.] \text{ then } \bot \text{ else } \text{True} \]
Q1: True \not\sim False?

One context suffices:
\[ C[.] = \text{if } [.] \text{ then } \bot \text{ else True} \]

Q2: mapStandard \sim mapWeird?

No: \((\lambda x. \bot) \downarrow\), but \(\bot \uparrow\)
Examples

Q1: True $\not\sim$ False?

One context suffices:
$C[.] = \text{if}[.\text{]}\text{ then } \bot \text{ else True}$

Q2: mapStandard $\sim$ mapWeird?

TODO: check infinitely many programs $P[.]$
whether $P[\text{mapStandard}] \downarrow \iff P[\text{mapWeird}] \downarrow$?
Examples

Q1: True \not\sim False?

One context suffices:
\[ C[.] = \text{if } [.] \text{ then } \bot \text{ else True} \]

Q2: mapStandard \sim mapWeird?

TODO: check infinitely many programs \( P[.] \)
whether \( P[\text{mapStandard}] \downarrow \iff P[\text{mapWeird}] \downarrow \)?

Q3: \( \lambda x. \bot \sim \bot \)?
Examples

Q1: True \not\sim False?

One context suffices:
\( C[.] = \text{if } [.] \text{ then } \bot \text{ else True} \)

Q2: mapStandard \sim mapWeird?

TODO: check infinitely many programs \( P[.] \)

whether \( P[\text{mapStandard}]\downarrow \iff P[\text{mapWeird}]\downarrow \)?

Q3: \( \lambda x.\bot \sim \bot \)?

No: \( (\lambda x.\bot)\downarrow \), but \( \bot \uparrow \)

Abramsky: The lazy lambda calculus, 1990
Remarks on Alternative Approaches
\[ [\cdot] : L \rightarrow D \]

- adequate: \[ [e_1] = [e_2] \implies e_1 \sim e_2 \]
\[ [\cdot] : L \rightarrow D \]

- adequate: \([e_1] = [e_2] \implies e_1 \sim e_2\)
- in general not fully abstract: \(e_1 \sim e_2\) but \([e_1] \neq [e_2]\) is possible.
  (usual argument: “parallel-or” is available in denotation, but not the language.)
Let $\to$ be the (compatible) reduction, i.e. permitted in all contexts. Let $\xrightarrow{s}$ be the (standard) reduction, i.e. under a strategy.

Definition: $\xrightarrow{s}$ is **standardizing**, iff

$$ e \xrightarrow{\ast} success \text{ implies } e \xrightarrow{s,\ast} success. $$

Proposition: If $\to$ is confluent, $\xrightarrow{s}$ is standardizing, and $\{success\}$ remains stable under reduction, then $\leftrightarrow$ is sound for contextual equivalence $\sim$.

However, in general $\leftrightarrow \subset \sim$ (i.e., $\sim$ is coarser than $\leftrightarrow$).

Confluence $= \implies$ determinism

In general $\xrightarrow{s}$ is nonterminating.
Let \( \rightarrow \) be the (compatible) reduction, i.e. permitted in all contexts.
Let \( s \rightarrow \) be the (standard) reduction, i.e. under a strategy.

**Definition:** \( s \rightarrow \) is **standardizing**, iff
\[
e \xrightarrow{*} \text{success} \implies e \xrightarrow{s,*} \text{success}.
\]

**Proposition** If \( \rightarrow \) is confluent, \( s \rightarrow \) is standardizing, and \( \{\text{success}\} \) remains stable under reduction, then \( \leftrightarrow \) is sound for contextual equivalence \( \sim \).
Let \( \rightarrow \) be the (compatible) reduction, i.e. permitted in all contexts.
Let \( S \rightarrow \) be the (standard) reduction, i.e. under a strategy.
Definition: \( S \rightarrow \) is standardizing, iff
\[
e \rightarrow^* \text{success} \text{ implies } e \stackrel{S,*}{\rightarrow} \text{success}.
\]

**Proposition** If \( \rightarrow \) is confluent, \( S \rightarrow \) is standardizing,
and \{success\} remains stable under reduction,
then \( \leftrightarrow^* \) is sound for contextual equivalence \( \sim \).

**However**
- In general \( \leftrightarrow^* \subset \sim \): \( \sim \) is coarser than \( \leftrightarrow^* \).
- Confluence \( \implies \) determinism
- In general \( S \rightarrow \) is nonterminating.
Diagrams and Correctness of Transformations

Calculus LR
LR  (core-language of Haskell)

- A pure (untyped) functional language with letrec, case, constructors, seq.
- Call-by-need (deterministic) reduction.
- Contextual equivalence based on may-convergence
Call-by-need reduction in LR (rules, a selection):

(lbeta) \((\lambda x. e_1) \ e_2 \rightarrow (\text{letrec}\ x = e_2\ \text{in}\ e_1)\)

(cp-in) \((\text{letrec}\ x_1 = v^S, \{x_i = x_{i-1}\}_{i=2}^m, Env\ \text{in}\ C[x_m^V]) \rightarrow (\text{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^m, Env\ \text{in}\ C[v]))\)

where \(v\) is an abstraction

(lllet) consists of two reduction rules:

(lllet-in) \((\text{letrec}\ Env_1\ \text{in}\ (\text{letrec}\ Env_2\ \text{in}\ r)^S) \rightarrow (\text{letrec}\ Env_1, Env_2\ \text{in}\ r)\)

(lllet-e) \((\text{letrec}\ Env_1, x = (\text{letrec}\ Env_2\ \text{in}\ s_x)^S\ \text{in}\ r) \rightarrow (\text{letrec}\ Env_1, Env_2, x = s_x\ \text{in}\ r)\)

S., Schütz, Sabel: Safety of Nöckers’s strictness analysis. JFP 2008
**Context Lemma in LR**

If for all reduction contexts $R$: $R[s] \downarrow \Rightarrow R[t] \downarrow$, then $s \leq_{\text{may}} t$.

Where reduction contexts are contexts around the redexes; (under the normal-order reduction strategy)
Context Lemmas

**Context Lemma in LR**

If for all reduction contexts $R$: $R[s] \downarrow \implies R[t] \downarrow$, then $s \leq_{may} t$.

Where reduction contexts are contexts around the redexes; (under the normal-order reduction strategy)

**Context Lemma in LR; a weaker variant; better suited for computing diagrams in LR**

If for all surface contexts $S$: $S[s] \downarrow \implies S[t] \downarrow$, then $s \leq_{may} t$

Where surface contexts are contexts where the hole is not in an abstraction.
Correctness Proofs using Diagrams

Forking diagrams for (llet) wrt. $S$-contexts; a complete set

\[
\begin{align*}
&\text{iS,llet} \\
&\begin{array}{c}
n,a \downarrow \\
iS,llet \downarrow \rightarrow \end{array} \\
&\begin{array}{c}
n,a \downarrow \\
iS,llet \downarrow \rightarrow \end{array} \\
&\begin{array}{c}
(n,lll)^+ \downarrow \\
iS,llet \downarrow \rightarrow \end{array} \\
&\begin{array}{c}
(n,lll)^+ \downarrow \\
iS,llet \downarrow \rightarrow \end{array}
\end{align*}
\]
Correctness Proofs using Diagrams

Forking diagrams for \((\text{llet})\)  

Purpose: a proof of \(\xrightarrow{\text{llet}} \subseteq \leq\).

\[
\begin{array}{c}
iS,\text{llet} \quad iS,\text{llet} \\
\downarrow n,a \quad \downarrow n,a \\
iS,\text{llet} \quad iS,\text{llet} \\
\downarrow n,a \\
iS,\text{llet} \\
\end{array}
\quad (n,\text{lll})+ \quad (n,\text{lll})+ \\
\downarrow \quad \downarrow \\
(n,\text{lll})+ \\
\quad (n,\text{lll})+ \\
\downarrow \\
iS,\text{llet} \\
\quad iS,\text{llet} \\
\downarrow n,a \\
\quad / \\
\quad / \\
\quad / n,a \\
\quad / \\
n,\text{llet} \\
\downarrow \\
\quad / \\
\quad / \\
\quad / \\
\quad / \\
\quad / \\
\end{array}
\]

Proof of \(e \Downarrow \land e \rightarrow e' \Rightarrow e' \Downarrow:\)
Correctness Proofs using Diagrams

Forking diagrams for (llet) Purpose: a proof of \( \text{llet} \subseteq \leq \).

Proof of \( e \downarrow \land e \xrightarrow{S, \text{llet},*} e' \implies e' \downarrow \):
Correctness Proofs using Diagrams

Forking diagrams for (llet) Purpose: a proof of $\xrightarrow{\text{llet}} \subseteq \leq$.

Proof of $e \downarrow \land e \xrightarrow{\text{S, llet,}^*} e' \implies e' \downarrow$:
Correctness Proofs using Diagrams

Forking diagrams for (llet) Purpose: a proof of \( \text{lllet} \subseteq \leq \).

\[
\begin{align*}
\text{iS, llet} & \quad \text{n, a} \quad \text{n, a} \\
\text{n, a} & \quad \text{n, a} \quad \text{n, a} \\
\text{n, a} & \quad \text{n, a} \quad \text{n, a} \\
\end{align*}
\]

\[
\begin{align*}
\text{iS, llet} & \quad \text{n, a} \quad \text{n, a} \\
\text{n, a} & \quad \text{n, a} \quad \text{n, a} \\
\text{n, a} & \quad \text{n, a} \quad \text{n, a} \\
\end{align*}
\]

Proof of \( e \downarrow \land e \xrightarrow{S, \text{llet}, *} e' \implies e' \downarrow \):

\[
\begin{align*}
e & \xrightarrow{\text{iS, llet}} e' \\
n & \downarrow \\
n, a & \downarrow \\
e \text{ WHNF} & \\
\end{align*}
\]

\[
\begin{align*}
e & \xrightarrow{\text{iS, llet}} e' \\
n & \downarrow \\
n, a & \downarrow \\
e \text{ WHNF} & \\
\end{align*}
\]

\[
\begin{align*}
e & \xrightarrow{\text{iS, llet}} e' \\
n & \downarrow \\
n, a & \downarrow \\
e \text{ WHNF} & \\
\end{align*}
\]
Correctness Proofs using Diagrams

Forking diagrams for (llet) Purpose: a proof of $\text{llet} \subseteq \leq$.

Proof of $e \downarrow \land e \xrightarrow{\text{S, llet},*} e' \implies e' \downarrow$:

\[
\begin{array}{cccc}
  e \xrightarrow{iS,\text{llet}} e' & e \xrightarrow{iS,\text{llet}} e' & e \xrightarrow{iS,\text{llet}} e' \\
  n \downarrow & n \downarrow & n \downarrow \\
  n,a \downarrow & n,a \downarrow & n,a \downarrow \\
  \cdot & \cdot & \cdot \\
  \cdot & \cdot & \cdot \\
  e_{\text{WHNF}} & e_{\text{WHNF}} & e_{\text{WHNF}} \\
\end{array}
\]
Correctness Proofs using Diagrams

Forking diagrams for (llet) Purpose: a proof of $\text{llet} \subseteq \leq$.

Proof of $e \downarrow \land e \xrightarrow{S, \text{llet}, \star} e' \implies e' \downarrow$: 
For the inverse direction
\[ e \downarrow \land e' \xrightarrow{*} e \implies e' \downarrow, \]
the method applies in a similar way:

- Commuting diagrams instead of forking diagrams.
Results

- A large set of correct program transformations

Applied in

Niehren, Sabel, S., Schwinghammer: Observational Semantics for a Concurrent Lambda Calculus with Reference Cells and Futures. ENTCS 2007

Sabel, S.: A call-by-need lambda calculus with locally bottom-avoiding choice. . . . MSCS 2008

Sabel, S.: A contextual semantics for concurrent Haskell with futures. PPDP 2011
Correctness Proofs using Diagrams

Results

- A large set of correct program transformations
- Several length measures of standard reductions (complexity of evaluations) and transformations that improve the complexity(ies).

Applied in

Niehren, Sabel, S., Schwinghammer: Observational Semantics for a Concurrent Lambda Calculus with Reference Cells and Futures. ENTCS 2007

Sabel, S.: A call-by-need lambda calculus with locally bottom-avoiding choice. . . . MSCS 2008

Sabel, S.: A contextual semantics for concurrent Haskell with futures. PPDP 2011
If the diagrams are known, and a correctness proof is required, then:

- The diagrams can be interpreted as meta-rewriting rules on reduction sequences consisting of standard reductions and transformations.
- The meta-irreducible reduction sequences are the standard reduction sequences.

(Rau, Sabel, S., IJCAR 2012)
If the diagrams are known, and a correctness proof is required, then:

- The diagrams can be interpreted as meta-rewriting rules on reduction sequences consisting of standard reductions and transformations.
- The meta-irreducible reduction sequences are the standard reduction sequences.
- It is sufficient to prove termination of the meta-rewriting system!
  (which was done (in LR) using termination provers AProVE (RWTH -Aachen) and IsaFoR / CeTA (University Innsbruck) )

(Rau, Sabel, S., IJCAR 2012)
Correctness Proofs using Diagrams

**Issue: Computation of Diagrams**

- Similar to computing critical pairs (ala Knuth-Bendix)
- Some extra complications:
  - higher-order and scoping
  - asymmetry due to reduction strategies
  - equational theories involved (in the letrec construct of Haskell)
  - ...

Automatic Computation of Diagrams

(C. Rau, thesis in preparation): Computing a complete set of forking diagrams for all reduction rules is decidable in LR (letrec, case, constructors,..) finitely many overlaps are sufficient.

Method: applying nominal unification techniques to LR
Correctness Proofs using Diagrams

**Issue: Computation of Diagrams**
- Similar to computing critical pairs (ala Knuth-Bendix)
- Some extra complications:
  - higher-order and scoping
  - asymmetry due to reduction strategies
  - equational theories involved (in the letrec construct of Haskell)
  - ...

**Automatic Computation of Diagrams**
(C. Rau, thesis in preparation):
- Computing a complete set of forking diagrams for all reduction rules is decidable in LR (letrec, case, constructors,..)
- finitely many overlaps are sufficient.
- Method: applying nominal unification techniques to LR
Concurrency, Non-Determinism and Contextual Semantics
May-Semantics: is perfect for a deterministic setting
**May-Semantics:** is perfect for a deterministic setting

But: has too low discriminating power for concurrent/nondeterministic evaluation

![Diagram](image)
**May-Semantics**: is perfect for a deterministic setting

**But**: has too low discriminating power for concurrent/nondeterministic evaluation

\[ P \sim P' \] by using only may-convergence
Contextual Semantics in Concurrency

\[ e_1 \leq_{\text{may}} e_2 \text{ iff } \forall C : C[e_1] \downarrow_{\text{may}} \implies C[e_2] \downarrow_{\text{may}} \]
\[ e_1 \sim_{\text{may}} e_2 \text{ iff } e_1 \leq_{\text{may}} e_2 \text{ and } e_2 \leq_{\text{may}} e_1 \]

\[ e_1 \leq_{\text{should}} e_2 \text{ iff } \forall C : C[e_1] \downarrow_{\text{should}} \implies C[e_2] \downarrow_{\text{should}} \]
\[ e_1 \sim_{\text{should}} e_2 \text{ iff } e_1 \leq_{\text{should}} e_2 \text{ and } e_2 \leq_{\text{should}} e_1 \]

\[ e_1 \sim e_2 \text{ iff } e_1 \sim_{\text{may}} e_2 \text{ and } e_1 \sim_{\text{should}} e_2 \]

Where: \( e \downarrow_{\text{should}} \) is defined as: \( \forall e' : e \xrightarrow{*} e' \implies e' \downarrow \)
Proposed by other researchers: may and must-convergence:

\[ e_1 \leq_{must} e_2 \text{ iff } \forall C : C[e_1] \downarrow_{must} \implies C[e_2] \downarrow_{must} \]

\[ e_1 \sim_{must} e_2 \text{ iff } e_1 \leq_{must} e_2 \text{ and } e_2 \leq_{must} e_1 \]

Where: \( e \downarrow_{must} \) is defined as:

every reduction sequence from \( e \) is successful (and terminating).
invariant properties: \(\text{Prop}(e) \land e \sim e' \implies \text{Prop}(e')\):

<table>
<thead>
<tr>
<th>invariances of (\sim)</th>
<th>may</th>
<th>may, should</th>
<th>may, must</th>
<th>may, should, must</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\exists) value</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>no error possible</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>no infinite reductions</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

**Discussion**

- more invariances mean less program transformations;
- fair evaluation must be explicitly required for \(\sim\)_{must};
- fair evaluation does not change \(\sim\)_{should};
- test-case: are “busy-wait-like”-implementations equivalent to buffer-implementations?
May, Should, Must-Convergence and Invariances

invariant properties: \( \text{Prop}(e) \land e \sim e' \implies \text{Prop}(e') \):

<table>
<thead>
<tr>
<th>invariances of ( \sim )</th>
<th>may</th>
<th>may, should</th>
<th>may, must</th>
<th>may, should, must</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \exists ) value</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>no error possible</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>no infinite reductions</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

**Proposal**

Use \( \sim_{\text{may}} \) and \( \sim_{\text{should}} \)
Concurrent Haskell with Futures (CHF):

Semantic analyses using contextual semantics
Concurrent Haskell with Futures (CHF):

- **Concurrent Haskell** (Peyton Jones, Gordon, Finne 1996) extends Haskell by concurrency

- The **process calculus CHF** (Sabel, S. CHF..., 2011) models **Concurrent Haskell with Futures** operational semantics inspired by (Peyton Jones, 2001)

- **Future** = variable whose value is computed concurrently by a monadic computation

- Futures allow **implicit synchronization by data dependency**

- Concurrent Haskell $+$ `unsafeInterleaveIO` can encode CHF (CHF is a sublanguage of Concurrent Haskell $+$ `unsafeInterleaveIO`
An **MVar** is a one-place buffer, which may be empty or filled.

- **takeMVar a** empties the **MVar** with address *a*
- **putMVar a** fills the (empty) **MVar** with address *a*
An **MVar** is a one-place buffer, which may be empty or filled.

- **takeMVar** *a* empties the **MVar** with address *a*
- **putMVar** *a* fills the (empty) **MVar** with address *a*
An **MVar** is a one-place **buffer**, which may be empty or filled.

- **takeMVar** \(a\) empties the **MVar** with address \(a\)
- **putMVar** \(a\) fills the (empty) **MVar** with address \(a\)
\[
x \leftarrow \text{main} \quad \text{takeMVar } a \mid z \leftarrow \text{putMVar } a \ (\text{length } u - 3) \gg \text{takeMVar } b \\
\mid y \leftarrow \text{putMVar } a \ 1 \mid u = [1, 2, 3, 4, 5] \mid a m - \mid b m 3
\]

\[
\text{\rightarrow}
\]

\[
x \leftarrow \text{main} \quad \text{takeMVar } a \mid z \leftarrow \text{takeMVar } b \\
\mid y \leftarrow \text{putMVar } a \ 1 \mid u = [1, 2, 3, 4, 5] \mid a m 2 \mid b m 3
\]

An \textit{MVar} is a \textbf{one-place buffer}, which may be empty or filled.

- \textit{takeMVar } a \text{} empties the \textit{MVar} with address \textit{a}
- \textit{putMVar } a \text{} fills the (empty) \textit{MVar} with address \textit{a}
\( x \leftarrow \text{main} \text{ takeMVar } a \mid z \leftarrow \text{putMVar } a \ (\text{length } u - 3) \gg \text{takeMVar } b \mid y \leftarrow \text{putMVar } a \ 1 \mid u = [1, 2, 3, 4, 5] \mid a m - \mid b m 3 \)

\[
\rightarrow
\]

\( x \leftarrow \text{main} \text{ takeMVar } a \mid z \leftarrow \text{takeMVar } b \mid y \leftarrow \text{putMVar } a \ 1 \mid u = [1, 2, 3, 4, 5] \mid a m 2 \mid b m 3 \)

\[
\rightarrow
\]

\( x \leftarrow \text{main} \text{ takeMVar } a \mid z = 3 \mid y \leftarrow \text{putMVar } a \ 1 \mid u = [1, 2, 3, 4, 5] \mid a m 2 \mid b m - \)

An \textit{MVar} is a one-place buffer, which may be empty or filled.

- \textit{takeMVar } a \text{ empties the MVar with address } a
- \textit{putMVar } a \text{ fills the (empty) MVar with address } a
An **MVar** is a one-place buffer, which may be empty or filled.

- **takeMVar a** empties the **MVar** with address **a**
- **putMVar a** fills the (empty) **MVar** with address **a**
\[
x \leftarrow \text{main} \quad \text{takeMVar } a \mid z \leftarrow \text{putMVar } a ((\text{length } u) - 3) \gg \text{takeMVar } b \\
\mid y \leftarrow \text{putMVar } a 1 \mid u = [1, 2, 3, 4, 5] \mid a m - \mid b m 3
\]
\[
\rightarrow
\]
\[
x \leftarrow \text{main} \quad \text{takeMVar } a \mid z \leftarrow \text{takeMVar } b \\
\mid y \leftarrow \text{putMVar } a 1 \mid u = [1, 2, 3, 4, 5] \mid a m 2 \mid b m 3
\]
\[
\rightarrow
\]
\[
x \leftarrow \text{main} \quad \text{takeMVar } a \mid z = 3 \\
\mid y \leftarrow \text{putMVar } a 1 \mid u = [1, 2, 3, 4, 5] \mid a m 2 \mid b m -
\]
\[
\rightarrow
\]
\[
x \leftarrow \text{main} \quad \text{return } 2 \mid z = 3 \\
\mid y \leftarrow \text{putMVar } a 1 \mid u = [1, 2, 3, 4, 5] \mid a m - \mid b m -
\]

An **MVar** is a one-place buffer, which may be empty or filled.

- **takeMVar** \(a\) empties the **MVar** with address \(a\)
- **putMVar** \(a\) fills the (empty) **MVar** with address \(a\)
The Process Calculus CHF

Processes: A list of components: \( P_1 | P_2 | \ldots | P_n \)

\( P, P_i \in \text{Proc} ::= P_1 | P_2 | \nu x. P | x \leftarrow e | x = e | x \text{ m e} | x \text{ m } \rightarrow \)

A process has a main thread: \( x \leftarrow e | P \)

Expressions

\( e, e_i \in \text{Expr}_{\text{CHF}} ::= x | \lambda x.e | (e_1 e_2) | \text{seq } e_1 e_2 | c \ e_1 \ldots e_{\alpha(c)} \)

| case \( T \) e of \( \ldots (c_{T,i} \ x_1 \ldots x_{\alpha(c_{T,i})} \rightarrow e_i) \ldots \)
| letrec \( x_1 = e_1 \ldots \ x_n = e_n \) in e
| return e | e_1 >>= e_2 | future e
| takeMVar e | newMVar e | putMVar e_1 e_2

Types: standard monomorphic type system
Operational Semantics: Reduction \( P_1 \xrightarrow{CHF} P_2 \)

- Small-step reduction \( \xrightarrow{CHF} \) (call-by-name variant)
  It is known that call-by-name and call-by-need are equivalent
  w.r.t. \( \sim \). (Sabel, S. CHF..., 2011)

- Rules are closed w.r.t. structural congruence and process contexts

- Reduction rules for monadic computation and functional evaluation

Examples for reduction rules:

**monadic:**

(\textsl{fork}) \( x \leftarrow M[\textbf{future} \ e] \xrightarrow{CHF} \nu y.(x \leftarrow M[\textbf{return} \ y] | y \leftarrow e), \ y \ \text{fresh} \)

**functional:**

(\textsl{beta}) \( y \leftarrow M[F[((\lambda x.e_1) \ e_2)]] \xrightarrow{CHF} y \leftarrow M[F[e_1[e_2/x]]] \)

evaluation contexts: \( E \); forcing contexts: \( F \); monadic contexts: \( M \);
CHF-Reduction is non-deterministic:

\[
\begin{align*}
  x \leftarrow \text{putMVar } a & \leftarrow 0 \\
  & | y \leftarrow \text{putMVar } a & \leftarrow 1 \\
  & | a \leftarrow m - \\
\end{align*}
\]

\[
\begin{align*}
  x \leftarrow \text{return unit} \\
  & | y \leftarrow \text{putMVar } a & \leftarrow 1 \\
  & | a \leftarrow m & \leftarrow 0
\end{align*}
\]
CHF: Successful Processes and Convergence

**Success**

A process $x \xleftarrow{\text{main}} \text{return } e | P$ is called successful

Note: $P$ may be reducible...

**May-convergence** $P \downarrow_{\text{may}}$ holds, whenever $P \xrightarrow{\text{CHF},*} P_{\text{success}}$.

**Should-convergence** $P \downarrow_{\text{should}}$ holds, whenever

$P \xrightarrow{\text{CHF},*} P' \implies P' \downarrow_{\text{may}}$.

$P_1 \leq_{\text{CHF,may}} P_2$, iff for all process contexts $\mathcal{D}$: $\mathcal{D}[P_1] \downarrow \implies \mathcal{D}[P_2] \downarrow$

$P_1 \leq_{\text{CHF,should}} P_2$, iff for all $\mathcal{D}$: $\mathcal{D}[P_1] \downarrow_{\text{should}} \implies \mathcal{D}[P_2] \downarrow_{\text{should}}$

$P_1 \sim P_2$, iff $P_1 \sim_{\text{CHF,may}} P_2$, and $P_1 \sim_{\text{CHF,should}} P_2$. 
Theorem: Every functional reduction is correct:

\[ P_1 \xrightarrow{\text{CHF, functional}} P_2 \text{ implies } P_1 \sim P_2. \]

Examples: beta-reduction, case-reduction, seq-reduction

Theorem: The monadic reductions (as standard reduction) with the exception of putMVar and takeMVar are correct.

Theorem: The monad laws for \( \gg= \) are correct, provided \((\text{seq } e_1 e_2)\) is only used for forcing functional expressions \( e_1 \).

Sabel, S.: A Contextual Semantics for CHF, PPDP 2011
**CHF Conservativity**

**Theorem** Embedding the pure functional part of CHF into full CHF is conservative.
Comparing $\sim_{CHF}$ and $\sim_{pure}$.

**Consequence of CHF Conservativity:**
Correct transformations (optimizations) in the pure functional part remain correct in CHF.

Sabel, S.: Conservative Concurrency in Haskell; LICS 2012
Overview of the Conservativity Proof

finite syntax

finite syntax

infinite trees

infinite trees

\( e_1 \sim_{c,CHF} e_2 \)

\( IT(e_1) \sim_{c,CHFI} IT(e_2) \)

\( e_1 \sim_{c,PF} e_2 \)

\( IT(e_1) \sim_{b,PFI} IT(e_2) \)
Non-Conservativity Results

Let \( CHFL = CHF + \text{lazy futures} \)

- lazy future =
  concurrent computation starts \textbf{only if} the value is \textbf{demanded}

- \( CHFL \) is \textbf{not a conservative extension} of \( PF \)

- Counterexample: \( \text{seq } e_2 \,(\text{seq } e_1 \,e_2) \sim_{PF} (\text{seq } e_1 \,e_2) \)

  But: \( \text{seq } e_2 \,(\text{seq } e_1 \,e_2) \not\sim_{CHFL} (\text{seq } e_1 \,e_2) \)

Since lazy futures are encodable with \texttt{unsafeInterleaveIO}:

- \( CHF + \texttt{unsafeInterleaveIO} \) is also \textbf{not a conservative extension} of \( PF \)
<table>
<thead>
<tr>
<th>Methods Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Forking and commutation diagrams.</td>
</tr>
<tr>
<td>• Context Lemmas for may- and should-convergence.</td>
</tr>
<tr>
<td>• Unfolding letrec into infinitary expressions and contextual equivalence</td>
</tr>
<tr>
<td>• Equivalence of call-by-need and call-by-name using infinitary expressions.</td>
</tr>
<tr>
<td>• Proving properties of translations (adequacy) and embeddings (full abstractness) (S., Niehren, Schwinghammer, Sabel: IFIP TCS 2008)</td>
</tr>
<tr>
<td>• Soundness and completeness of applicative similarity (for may-convergence) for infinitary pure expressions, using Howe’s method.</td>
</tr>
</tbody>
</table>
Proving Correctness of a Concurrent Implementation of Software Transactional Memory
Two program calculi for STM Haskell: (S., Sabel, ICFP 2013)

\[ \text{SHF} \quad \xrightarrow{\text{translation } \psi} \quad \text{CSHF} \]

Definition of Correctness:
The implementation fulfills the specification

\[ P \downarrow \text{may} \iff \psi(P) \downarrow \text{may} \]
\[ P \downarrow \text{sh} \iff \psi(P) \downarrow \text{sh} \]

means: CSHF is a correct evaluator of SHF.

more general and more abstract:
\[ \psi(e_1) \leq \psi(e_2) \iff e_1 \leq e_2 \]
Two program calculi for STM Haskell: (S., Sabel, ICFP 2013)

\[ \text{Definition of Correctness:} \]

The implementation fulfills the specification

\[ P \downarrow_{\text{may}} \iff \psi(P) \downarrow_{\text{may}} \text{ and } P \downarrow_{\text{sh}} \iff \psi(P) \downarrow_{\text{sh}} \]

means: CSHF is a correct evaluator of SHF.
Two program calculi for STM Haskell: (S., Sabel, ICFP 2013)

\[ SHF \xrightarrow{\text{translation } \psi} CSHF \]

### Specification

**Definition of Correctness:**

The implementation fulfills the specification

\[ P \downarrow_{may} \iff \psi(P) \downarrow_{may} \quad \text{and} \quad P \downarrow_{sh} \iff \psi(P) \downarrow_{sh} \]

means: CSHF is a correct evaluator of SHF.

more general and more abstract:

\[ \psi \text{ is semantics reflecting: } \psi(e_1) \leq \psi(e_2) \implies e_1 \leq e_2 \]
Software Transactional Memory (STM)

- treats **shared memory** operations as **transactions**
- provides **lock-free** and **very convenient** concurrent programming
- requires an **implementation** that **correctly executes** the transactions
STM Haskell

- STM library for Haskell
- introduced by Harris et.al, PPoPP’05
- uses Haskell’s **strong type system** to distinguish between
  - IO-computations, (IO-monad)
  - software transactions, and (STM-monad)
  - functional code
Transactional Variables:

\textbf{TVar} \ a

Primitives to form STM-transactions \textbf{STM} \ a:

newTVar \quad \textbf{readTVar} \quad \textbf{writeTVar}

retry \ e \quad \textbf{orElse} \ e \ e'

Executing an STM-transaction:

atomically \ e

\textbf{Semantics}: the transaction-execution is

\begin{itemize}
  \item \textbf{atomic}: all or nothing, effects are indivisible, and
  \item \textbf{isolated}: concurrent evaluation is not observable
\end{itemize}
Specification: Process Calculus $SHF$

$SHF$ is a process calculus similar to CHF

- top-level: processes, futures and the TVars
- second level: are transactions (STM-monad)
- third level: the functional evaluation:
  extended lambda calculus with case, constructors, letrec
Operational Semantics:

- **Call-by-need** “small-step” reduction $\xrightarrow{\text{SHF}}$, ...

- **Big-step rule** for transactional evaluation:

  $$\begin{align*}
  &\mathbb{D}_1[\langle u \bowtie y \rangle \leftarrow M[\text{atomically } e]] \xrightarrow{\text{SHFA,}*} \mathbb{D}'_1[\langle u \bowtie y \rangle \leftarrow M[\text{atomically } (\text{return}_{\text{STM}} e')]] \\
  &\mathbb{D}[\langle u \bowtie y \rangle \leftarrow M[\text{atomically } e]] \xrightarrow{\text{SHF}} \mathbb{D}'[\langle u \bowtie y \rangle \leftarrow M[\text{return}_{\text{IO}} e']] \\
  \end{align*}$$

  where $\xrightarrow{\text{SHFA}}$ are small-step rules for transactional evaluation

  Informally: If the transaction can be completely executed while all other threads are stopped, then execute it.

- Enforces **sequential and atomic evaluation** of transactions in the specification calculus $\text{SHF} \Rightarrow$ atomicity and isolation obviously hold
Specification: Process Calculus *SHF*

**Operational Semantics:**

- **Call-by-need** “small-step” reduction $\xrightarrow{SHF}, \ldots$

- **Big-step rule** for transactional evaluation:

  $$
  \begin{align*}
  \mathcal{D}_1[\langle uy \rangle \leftarrow M[\text{atomically } e]] & \xrightarrow{SHFA,*} \mathcal{D}_1'[\langle uy \rangle \leftarrow M[\text{atomically } (\text{return}_{STM} e')]]) \\
  \mathcal{D}[\langle uy \rangle \leftarrow M[\text{atomically } e]] & \xrightarrow{SHF} \mathcal{D}'[\langle uy \rangle \leftarrow M[\text{return}_{IO} e']] \\
  \end{align*}
  $$

  where $\xrightarrow{SHFA}$ are small-step rules for transactional evaluation

  Informally: If the transaction can be completely executed while all other threads are stopped, then execute it.

- Enforces **sequential and atomic evaluation** of transactions in the specification calculus $SHF \Rightarrow$ atomicity and isolation obviously hold

- Whether a transaction can be executed is **undecidable** in the $SHF$-operational semantics.
Concurrent Implementation: Calculus $CSHF$

Extensions w.r.t. $SHF$:

- **local** copies of the used **global** TVars:
- Bookkeeping per thread of read and written TVars... is a stack due to nested orElse-s
- Bookkeeping of potentially conflicting threads (at the TVars)
A CSHF-example rule:

**Read-operation (readg):** A read first looks into the local store. If no local TVar exists, then the global value is copied into the local store and the own thread identifier is added to the notify-list of the global TVar.

(readg):

$$\langle u \wr y \rangle \xleftarrow{T,L} M_{\text{STM}}[\text{readTVar } x] | x \text{tg } e_1 - g | u \text{tls } r : s$$

$$\xrightarrow{\text{CSHF}} \nu z.\langle u \wr y \rangle \xleftarrow{T',L'} M_{\text{STM}}[\text{returnSTM } z] | z = e_1$$

$$| u \text{tls } (\{x \text{tl } z\} \cup r) : s | x \text{tg } z - g'$$

if $x \notin L_a$ where $L = (L_a, L_n, L_w) : L_r$,

$L' = (L_a \cup \{x\}, L_n, L_w) : L_r$,

$T' = T \cup \{x\}$ and $g' = (\{u\} \cup g)$
Operational semantics:

- **true small-step** reduction $\xrightarrow{\text{CSHF}}$
- **concurrent** evaluation of threads and also concurrent evaluation of STM transactions
- all rule applications are **decidable**

Transaction execution (informally):

- all writes are performed on **local** TVars
- The read and written TVars (only the names) are logged
- Bookkeeping of **notify list** of threads at TVars.

Commit phase

1. **lock** (relevant) global TVars
2. **send a retry** to all threads in the **notify lists** of to-be-written TVars (\(\equiv\) conflicting threads)
3. write content of local TVars into global TVars
4. remove the locks
Correctness of the Implementation

**Main Theorem**

**Convergence Equivalence:** For any $SHF$-process $P$:

$$P \downarrow_{SHF} \iff \psi(P) \downarrow_{CSHF} \quad \text{and} \quad P \downarrow_{SHF} \iff \psi(P) \downarrow_{CSHF}$$

**Adequacy:** For all $P_1, P_2 \in SHF$:

$$\psi(P_1) \sim_{CSHF} \psi(P_2) \implies P_1 \sim_{SHF} P_2$$
Proof methods

- commutation/non-commutation of reduction steps in reduction sequences.
- This requires four partial proofs for convergence equivalence.

- Analyzing and exploiting properties of translations: compositionality and convergence equivalence.
Proof methods

- commutation/non-commutation of reduction steps in reduction sequences.
- This requires four partial proofs for convergence equivalence.
  - $p \downarrow \implies \Psi(p) \downarrow$
  - $\Psi(p) \downarrow \implies p \downarrow$
  - $p \uparrow \implies \Psi(p) \uparrow$ (Base-case already covered)
  - $\Psi(p) \uparrow \implies p \uparrow$

- Analyzing and exploiting properties of translations: compositionality and convergence equivalence.
Consequences of Correctness

⇒ CSHF is a **correct evaluator** for SHF
⇒ Correct **program transformations** in CSHF are also correct for SHF

Consequences of correctness and of the proof

- Every (successful) reduction sequence in the specification SHF is also possible in the implementation CSHF.
- Every (successful) reduction sequence in the implementation CSHF can be retranslated into a successful reduction sequence in the specification calculus SHF.
- A progress property of the implementation: success of at least one of several conflicting transactions
Contextual semantics can be applied to deterministic as well as non-deterministic and also concurrent programming languages.

Requirement is only: syntax of expressions and contexts, an operational semantics, and definition of values.

The (theoretical and practical) tools have increased their power:
context lemmata, applicative (bi)simulations, diagrams, translations, combinations of may and should

Rewriting techniques can be applied to small-step operational semantics

Large examples are within reach of the methods of contextual semantics (cf. STM correctness).

A drawback:
Reasoning is tedious and often too syntactical
Future Work

**Future work; more work required:**

- Polymorphic typing and contextual equivalence
- More applicative (bi-)simulations also for concurrency (wrt. should-convergence)
- Complexity of reduction sequences
- Deeper analysis of translations
- Invariances of contextual preorder and equivalences
- Automating the operational reasoning
- ...