A Contextual Semantics for Concurrent Haskell with Futures

David Sabel & Manfred Schmidt-Schauß
Goethe-University, Frankfurt am Main, Germany

PPDP’11, Odense, Denmark
Motivation

- **Haskell**’s monadic IO allows a clean separation of pure functional expressions and side-effects

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

- We propose to extend Concurrent Haskell by concurrent futures to obtain a more declarative programming style for concurrency

- Our language model: process calculus CHF inspired by (Peyton Jones, 2001) and (Niehren et. al. 2006)
Is Concurrent Haskell with Futures “semantically sound”?

- **Correctness** of compiler optimizations and program transformations
- Do **monad laws** hold?
- Requires a notion of **program equivalence**
**Futures**

**Future** = Variable whose value becomes available in the future

We consider **concurrent, imperative, implicit** futures:

- **concurrent**: the value is computed by a concurrent thread
- **imperative**: the value is obtained by a monadic computation in the IO-monad.
- **implicit**: threads implicitly block until the demanded value of a future is available, no explicit force required

Declarative style:
Implicit futures allow **implicit synchronisation by data dependency**

Example:  
```haskell
do  
x1 ← future e1  
x2 ← future e2  
print (x1 + x2)
```
Concurrent Haskell

Concurrent Haskell = Haskell + threads + MVars (synchronizing variables)

- Thread creation: \( \text{forkIO} :: \text{IO} \ a \to \text{IO} \ \text{ThreadId} \)
- MVar creation: \( \text{newMVar} :: \text{a} \to \text{IO} \ (\text{MVar} \ a) \)
- Reading a filled MVar: \( \text{takeMVar} :: \text{MVar} \ a \to \text{IO} \ \text{a} \)
- Writing into an empty MVar: \( \text{putMVar} :: \text{MVar} \ a \to \text{a} \to \text{IO} () \)

Encoding implicit futures in Concurrent Haskell using lazy IO:

\[
\text{future} :: \text{IO} \ a \to \text{IO} \ a
\]

\[
\text{future act} = \text{do} \ \text{ack} \leftarrow \text{newEmptyMVar}
\]

\[
\text{thread} \leftarrow \text{forkIO} \ (\text{act} >>= \text{putMVar} \ \text{ack})
\]

\[
\text{unsafeInterleaveIO} \ (\text{takeMVar} \ \text{ack})
\]
The Process Calculus CHF: Syntax

Processes

\[ P, P_i \in \text{Proc} ::= P_1 | P_2 \quad \text{(parallel composition)} \]
\[ \nu x.P \quad \text{(name restriction)} \]
\[ x \leftarrow e \quad \text{(concurrent thread, future } x) \]
\[ x = e \quad \text{(binding)} \]
\[ x \_m \quad \text{(filled MVar)} \]
\[ x \_m \quad \text{(empty MVar)} \]

A process has a main thread: \[ x \leftarrow e \quad \text{main thread} \]

Expressions & Monadic Expressions

\[ e, e_i \in \text{Expr} ::= me | x | \lambda x.e | (e_1 e_2) | \text{seq } e_1 e_2 | c e_1 \ldots e_{\text{ar}(c)} \]
\[ \text{case } T e \text{ of } ... (c_{T,i} x_1 \ldots x_{\text{ar}(c_{T,i})} \rightarrow e_i ) ... \]
\[ \text{letrec } x_1 = e_1 \ldots x_n = e_n \text{ in } e \]

\[ me \in \text{MExpr} ::= \text{return } e | e_1 \triangleright= e_2 | \text{future } e \]
\[ \quad \text{takeMVar } e | \text{newMVar } e | \text{putMVar } e_1 e_2 \]
The Process Calculus CHF: Syntax

**Processes**

\[ P, P_i \in \text{Proc} ::= P_1 | P_2 \] (parallel composition)

\[ \nu x. P \] (name restriction)

\[ x \leftarrow e \] (concurrent thread, future \( x \))

\[ x = e \] (binding)

\[ x \mathbf{m} e \] (filled MVar)

\[ x \mathbf{m} - \] (empty MVar)

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

**Expressions & Monadic Expressions**

\[ e, e_i \in \text{Expr} ::= me | x | \lambda x. e | (e_1 e_2) | \text{seq } e_1 e_2 | c e_1 \ldots e_{\text{ar}(c)} \]

\[ \text{case}_T e \text{ of } \ldots (c_{T,i} x_1 \ldots x_{\text{ar}(c_{T,i})} \rightarrow e_i) \ldots \]

\[ \text{letrec } x_1 = e_1 \ldots x_n = e_n \text{ in } e \]

\[ me \in \text{MExpr} ::= \text{return } e | e_1 \gg e_2 | \text{future } e \]

\[ \text{takeMVar } e | \text{newMVar } e | \text{putMVar } e_1 e_2 \]
The Process Calculus CHF: Typing

Syntax of (monomorphic) types

\[ \tau, \tau_i \in Typ ::= (T \tau_1 \ldots \tau_n) \mid \tau_1 \to \tau_2 \mid IO \tau \mid MVar \tau \]

Type system:

- Usual monomorphic type system with recursive data constructors
- An exception is \( \text{seq} :: \tau_1 \to \tau_2 \to \tau_2 \)
  \( \tau_1 \) must not be an IO- or MVar-type

Otherwise, the monad laws would not hold even in usual Haskell!

Example: left unit law: \((\text{return } e_1) >>= e_2 \neq (e_2 \ e_1)\)

Prelude> seq ((return True >>= undefined)::IO ()) True
True
Prelude> seq ((undefined True)::IO ()) True
*** Exception: Prelude.undefined
Operational Semantics

Structural congruence \(\equiv\)  (similar as in the \(\pi\)-calculus)

\[
\begin{align*}
P_1 | P_2 & \equiv P_2 | P_1 \\
(P_1 | P_2) | P_3 & \equiv P_1 | (P_2 | P_3) \\
(\nu x. P_1) | P_2 & \equiv \nu x. (P_1 | P_2), \text{ if } x \notin \text{FV}(P_2) \\
\nu x_1. \nu x_2. P & \equiv \nu x_2. \nu x_1. P \\
P_1 & \equiv P_2, \text{ if } P_1 =_\alpha P_2 \\
D[P_1] & \equiv D[P_2], \text{ if } P_1 \equiv P_2, D \text{ a process context}
\end{align*}
\]

Process contexts: \(D ::= [\cdot] \mid D \mid P \mid P \mid D \mid \nu x. D\)

Operational Semantics: Reduction \(P_1 \xrightarrow{sr} P_2\)

- Small-step reduction
- Rules are closed w.r.t. \(\equiv\) and \(D\)-contexts
- Reduction rules for monadic computation and functional evaluation

David Sabel  Concurrent Haskell with Futures  8/19
Rules for Monadic Computations

- performed inside monadic contexts: \( M ::= \cdot | M >>= e \)
- direct implementation of the monad:
  \[
  \text{(lunit)} \quad x \leftarrow M[\text{return } e_1 >>= e_2] \xrightarrow{sr} x \leftarrow M[e_2 e_1]
  \]
- future creation:
  \[
  \text{(fork)} \quad x \leftarrow M[\text{future } e] \xrightarrow{sr} \nu y. (x \leftarrow M[\text{return } y] | y \leftarrow e), \ y \text{ fresh}
  \]
- completed evaluation of a future:
  \[
  \text{(unIO)} \quad y \leftarrow \text{return } e \xrightarrow{sr} y = e, \text{ if the thread is not the main-thread}
  \]
- operations on MVars:
  \[
  \text{(nmvar)} \quad y \leftarrow M[\text{newMVar } e] \xrightarrow{sr} \nu x. (y \leftarrow M[\text{return } x] | x \text{ m } e)
  \]
  \[
  \text{(tmvar)} \quad y \leftarrow M[\text{takeMVar } x] | x \text{ m } e \xrightarrow{sr} y \leftarrow M[\text{return } e] | x \text{ m }-
  \]
  \[
  \text{(pmvar)} \quad y \leftarrow M[\text{putMVar } x \ e] | x \text{ m }- \xrightarrow{sr} y \leftarrow M[\text{return } ()] | x \text{ m } e
  \]

David Sabel  Concurrent Haskell with Futures 9/19
Rules for Functional Evaluation

Functional evaluation performs call-by-need evaluation with sharing

- Sharing $\beta$-reduction:
  \[(l\beta) \ L[((\lambda x.e_1) e_2)] \xrightarrow{sr} \nu x. (L[e_1] \mid x = e_2)\]

- Copying abstractions & variables:
  \[(cp) \ \hat{L}[x] \mid x = v \xrightarrow{sr} \hat{L}[v] \mid x = v, \ v \text{ an abstraction or a variable}\]

- Further rules for copying constructors, case- and seq-reduction, and letrec

- Monadic operators are treated like constructors

$L$-contexts: $L ::= x \gets M[F]$

- $x \gets M[F[x_n]] \mid x_n = E_n[x_{n-1}] \mid \ldots \mid x_2 = E_2[x_1] \mid x_1 = E_1$

Evaluation contexts: $E ::= [\cdot] \mid (E \ e) \mid (\text{case } E \text{ of } alts) \mid (\text{seq } E \ e)$

Forcing contexts: $F ::= E \mid (\text{takeMVar } E) \mid (\text{putMVar } E \ e)$
Process $P$ is **successful** if

\[ P \text{ well-formed } \land P \equiv \nu x_i(x \xrightarrow{\text{main}} \text{return } e \mid P') \]

**May-Convergence**: (a successful process can be reached by reduction)

\[ P \downarrow \text{ iff } P \text{ is w.-f. and } \exists P' : P \xrightarrow{\text{sr},*} P' \land P' \text{ successful} \]

**Should-Convergence**: (every successor is may-convergent)

\[ P \downarrow \uparrow \text{ iff } P \text{ is w.-f. and } \forall P' : P \xrightarrow{\text{sr},*} P' \implies P' \downarrow \]

**Contextual Equivalence**

\[ P_1 \sim_c P_2 \text{ iff } \forall D : (D[P_1] \downarrow \iff D[P_2] \downarrow) \land (D[P_1] \downarrow \iff D[P_2] \downarrow) \]

Analogous on expressions $e_i$ of type $\tau$: $e_1 \leq_{c,\tau} e_2$ and $e_1 \sim_{c,\tau} e_2$. 
Fairness

Proposition

\[ \downarrow, \downarrow, \leq_c, \sim_c \text{ do not change} \]

if only **fair reduction sequences** are allowed

An **infinite** reduction sequence \( P_1 \xrightarrow{sr} P_2 \xrightarrow{sr} \ldots \) is **unfair** if

\[ P_1 \xrightarrow{sr} P_2 \xrightarrow{sr} \ldots \text{ has an infinite suffix } P_j \xrightarrow{sr} P_{j+1} \xrightarrow{sr} \ldots \]

where a (reducible) thread is never reduced
A proof tool to show equivalences:

**Context Lemma for Expressions**

If $\forall D[L[\cdot, \tau]]$-contexts:

$$D[L[e_1]] \downarrow \iff D[L[e_2]] \downarrow \text{ and } D[L[e_1]] \downarrow \iff D[L[e_2]] \downarrow$$

Then $e_1 \sim_{c, \tau} e_2$. 
Results: Call-by-name Evaluation is Correct

Call-by-name Reduction

Small-step reduction $\xrightarrow{src}$ with full substitution, no sharing:

(cpce) \[ y \leftarrow M[F[x]] \mid x = e \xrightarrow{src} y \leftarrow M[F[e]] \mid x = e \]

(nbeta) \[ y \leftarrow M[F[((\lambda x.e_1) e_2)]] \xrightarrow{src} y \leftarrow M[F[e_1[e_2/x]]] \]

(ncase) \[ y \leftarrow M[F[\text{case}_T (c e_1 \ldots e_n) \text{ of } \ldots ((c y_1 \ldots y_n) \rightarrow e) \ldots]] \xrightarrow{src} y \leftarrow M[F[e[e_1/y_1, \ldots, e_n/y_n]]] \]

$\downarrow_{src}, \downarrow_{-src}$: call-by-name may- & should-convergence

Theorem

\[ P \downarrow \iff P \downarrow_{src} \text{ and } P \downarrow \iff P \downarrow_{src} \]
Outline of the Proof

Translation $IT :: \text{CHF} \rightarrow \text{CHFI}$ unfolds all bindings into infinite trees, e.g.

$$IT \left( \text{letrec } xs = (\text{True} : xs) \ \text{in } xs \right) = \text{True}$$

Steps of the proof

- CHFI = calculus with infinite trees, no letrec, no bindings
- Call-by-name reduction on infinite trees
- Convergence equivalence: tree reduction and call-by-need reduction
- Convergence equivalence: tree reduction and call-by-name reduction
Correct Program Transformations

**Correctness**

A transformation on processes \( P_1 \rightarrow P_2 \) is correct iff \( P_1 \sim_c P_2 \)

A transformation on expressions \( e_1 \rightarrow e_2 \) is correct iff \( e_1 \sim_{c,\tau} e_2 \)

**Results on Reductions**

- All rules for functional evaluation are correct in any context
- \((sr, \text{lunit}), (sr, \text{nmvar}), (sr, \text{fork}), (\text{unIO})\) are correct
- \((sr, \text{tmvar})\) and \((sr, \text{pmvar})\) are in general not correct
- Deterministic take and put are correct:

\[
\nu x. D[y \leftarrow M[\text{takeMVar } x] \mid x \text{ m e}] \rightarrow \nu x. D[y \leftarrow M[\text{return } e] \mid x \text{ m –]}
\]

if no other \text{takeMVar} on \(x\) is possible in any context

\[
\nu x. D[y \leftarrow M[\text{putMVar } x e] \mid x \text{ m –}] \rightarrow \nu x. D[y \leftarrow M[\text{return } ()] \mid x \text{ m e]}
\]

if no other \text{putMVar} on \(x\) is possible in any context
Results on other transformations

- General copying (gcp):
  \[
  (\text{gcp}) \quad C[x] \mid x = e \rightarrow C[e] \mid x = e
  \]

- Garbage collection (gc):
  \[
  (\text{gc}) \quad \nu x_1, \ldots, x_n. (P \mid \text{Comp}(x_1) \mid \ldots \mid \text{Comp}(x_n)) \rightarrow P
  \]
  where every \(\text{Comp}(x_i)\) is
  - a binding \(x_i = e_i\),
  - an MVar \(x_i \mathbf{m} e_i\), or
  - an empty MVar \(x_i \mathbf{m} \leftarrow\)
  and \(x_i \not\in FV(P)\).
Monad Laws

Theorem

The monad laws

\begin{align*}
(M1) & \quad \text{return } e_1 \gg= e_2 \quad \sim_c e_2 e_1 \\
(M2) & \quad e_1 \gg= \lambda x.\text{return } x \quad \sim_c e_1 \\
(M3) & \quad e_1 \gg= (\lambda x.(e_2 x \gg= e_3)) \quad \sim_c (e_1 \gg= e_2) \gg= e_3
\end{align*}

are correct.

⇒ use of do-notation is correct

\begin{verbatim}
do
  x1 ← future e1
  x2 ← future e2
return (x1 + x2)
\end{verbatim}
Conclusion

- CHF models Concurrent Haskell with futures
- Contextual equivalence based on may- and should-convergence
- Call-by-need and call-by-name are equivalent in CHF
- A lot of program transformations are correct
- The monad laws hold, but the type of seq must be restricted
- do-notation is available

Further Work

- Is CHF referentially transparent?
- Analyze further extensions:
  - Exceptions
  - killThread
  - ...