# A Contextual Semantics for Concurrent Haskell with Futures

**David Sabel & Manfred Schmidt-Schauß**

Goethe-University, Frankfurt am Main, Germany

PPDP'11, Odense, Denmark

# Motivation

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

- **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)

# Issues

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

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:

```
do
  x1 ← future e1
  x2 ← future e2
  print (x1 + x2)
```

# Concurrent Haskell

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

- Thread creation: `forkIO :: IO a → IO ThreadId`
- MVar creation: `newMVar :: a → IO (MVar a)`
- Reading a filled MVar: `takeMVar :: MVar a → IO a`
- Writing into an empty MVar: `putMVar :: MVar a → a → IO ()`

Encoding implicit futures in Concurrent Haskell using lazy IO:

```
future :: IO a → IO a
future act = do ack ←newEmptyMVar
                thread ←forkIO (act >>= putMVar ack)
                unsafeInterleaveIO (takeMVar ack)
```

# The Process Calculus CHF: Syntax

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

## Processes

$$
\begin{array}{llll}
P, P_i \in \mathit{Proc} & ::= & P_1 \mid P_2 & \text{(parallel composition)} \\
 & \mid & \nu x.P & \text{(name restriction)} \\
 & \mid & x \Leftarrow e & \text{(concurrent thread, future } x\text{)} \\
 & \mid & x = e & \text{(binding)} \\
 & \mid & x\,\mathbf{m}\,e & \text{(filled MVar)} \\
 & \mid & x\,\mathbf{m}\,- & \text{(empty MVar)}
\end{array}
$$

A process has a main thread: $x \stackrel{\text{main}}{\Longleftarrow} e \mid P$

## Expressions & Monadic Expressions

$$
\begin{array}{lll}
e, e_i \in \mathit{Expr} ::= & me \mid x \mid \lambda x.e \mid (e_1\ e_2) \mid \mathtt{seq}\ e_1\ e_2 \mid c\ e_1 \ldots e_{\mathrm{ar}(c)} \\
 & \mid \mathtt{case}_T\ e\ \mathtt{of}\ \ldots (c_{T,i}\ x_1 \ldots x_{\mathrm{ar}(c_{T,i})} \to e_i) \ldots \\
 & \mid \mathtt{letrec}\ x_1 = e_1\ \ldots\ x_n = e_n\ \mathtt{in}\ e
\end{array}
$$

$$
\begin{array}{ll}
me \in \mathit{MExpr} ::= & \mathtt{return}\ e \mid e_1 \gg= e_2 \mid \mathtt{future}\ e \\
 & \mid \mathtt{takeMVar}\ e \mid \mathtt{newMVar}\ e \mid \mathtt{putMVar}\ e_1\ e_2
\end{array}
$$

# The Process Calculus CHF: Syntax

## Processes

$$
\begin{array}{rcll}
P, P_i \in \mathit{Proc} & ::= & P_1 \mid P_2 & \text{(parallel composition)} \\
& \mid & \nu x.P & \text{(name restriction)} \\
& \mid & x \Leftarrow e & \text{(concurrent thread, future } x\text{)} \\
& \mid & x = e & \text{(binding)} \\
& \mid & x\,\mathbf{m}\,e & \text{(filled MVar)} \\
& \mid & x\,\mathbf{m}\,- & \text{(empty MVar)}
\end{array}
$$

A process has a main thread: $x \stackrel{\text{main}}{\Longleftarrow} e \mid P$

## Expressions & Monadic Expressions

$$
\begin{array}{rl}
e, e_i \in \mathit{Expr} ::= & me \mid x \mid \lambda x.e \mid (e_1\ e_2) \mid \texttt{seq}\ e_1\ e_2 \mid c\ e_1 \ldots e_{\mathrm{ar}(c)} \\
& \mid \texttt{case}_T\ e\ \texttt{of}\ \ldots (c_{T,i}\ x_1 \ldots x_{\mathrm{ar}(c_{T,i})} \rightarrow e_i) \ldots \\
& \mid \texttt{letrec}\ x_1 = e_1\ \ldots\ x_n = e_n\ \texttt{in}\ e
\end{array}
$$

$$
\begin{array}{rl}
me \in \mathit{MExpr} ::= & \texttt{return}\ e \mid e_1 \gg\!= e_2 \mid \texttt{future}\ e \\
& \mid \texttt{takeMVar}\ e \mid \texttt{newMVar}\ e \mid \texttt{putMVar}\ e_1\ e_2
\end{array}
$$

# The Process Calculus CHF: Typing

**Syntax of (monomorphic) types**

$$\tau, \tau_i \in \mathit{Typ} ::= (T\ \tau_1\ \ldots\ \tau_n) \mid \tau_1 \to \tau_2 \mid \mathtt{IO}\ \tau \mid \mathtt{MVar}\ \tau$$

Type system:

- Usual monomorphic type system with recursive data constructors
- An exception is 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: $(\mathtt{return}\ e_1) \ggg\!= e_2 \quad \neq \quad (e_2\ e_1)$

```
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)*

$$
\begin{aligned}
P_1 \mid P_2 &\equiv P_2 \mid P_1\\
(P_1 \mid P_2) \mid P_3 &\equiv P_1 \mid (P_2 \mid P_3)\\
(\nu x.P_1) \mid P_2 &\equiv \nu x.(P_1 \mid P_2),\ \textit{if } x \notin FV(P_2)\\
\nu x_1.\nu x_2.P &\equiv \nu x_2.\nu x_1.P\\
P_1 &\equiv P_2,\ \textit{if } P_1 =_\alpha P_2\\
\mathbb{D}[P_1] &\equiv \mathbb{D}[P_2],\ \textit{if } P_1 \equiv P_2,\ \mathbb{D} \textit{ a process context}
\end{aligned}
$$

*Process contexts:* $\mathbb{D} ::= [\cdot] \mid \mathbb{D} \mid P \mid P \mid \mathbb{D} \mid \nu x.\mathbb{D}$

**Operational Semantics: Reduction $P_1 \xrightarrow{sr} P_2$**

- *Small-step reduction*
- *Rules are closed w.r.t. $\equiv$ and $\mathbb{D}$-contexts*
- *Reduction rules for monadic computation and functional evaluation*

# Rules for Monadic Computations

- performed inside monadic contexts: $\mathbb{M} ::= [\cdot] \mid \mathbb{M} \gg\!= e$
- direct implementation of the monad:

  (lunit) $x \Leftarrow \mathbb{M}[\mathtt{return}\ e_1 \gg\!=\ e_2] \xrightarrow{sr} x \Leftarrow \mathbb{M}[e_2\ e_1]$
- future creation:

  (fork) $x \Leftarrow \mathbb{M}[\mathtt{future}\ e] \xrightarrow{sr} \nu y.(x \Leftarrow \mathbb{M}[\mathtt{return}\ y] \mid y \Leftarrow e)$, $y$ fresh
- completed evaluation of a future:

  (unIO) $y \Leftarrow \mathtt{return}\ e \xrightarrow{sr} y = e$, if the thread is not the main-thread
- operations on MVars:

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

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

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

# Rules for Functional Evaluation

Functional evaluation performs call-by-need evaluation with sharing

- Sharing $\beta$-reduction:

  (lbeta) $\mathbb{L}[((\lambda x.e_1)\ e_2)] \xrightarrow{sr} \nu x.(\mathbb{L}[e_1] \mid x = e_2)$

- Copying abstractions & variables:

  (cp) $\widehat{\mathbb{L}}[x] \mid x = v \xrightarrow{sr} \widehat{\mathbb{L}}[v] \mid x = v,\quad v$ an abstraction or a variable

- further rules for copying constructors, `case`- and `seq`-reduction, and `letrec`
- monadic operators are treated like constructors

$\mathbb{L}$-contexts: $\mathbb{L} ::= x \Leftarrow \mathbb{M}[\mathbb{F}]$
$\mid\ x \Leftarrow \mathbb{M}[\mathbb{F}[x_n]] \mid x_n = \mathbb{E}_n[x_{n-1}] \mid \ldots \mid x_2 = \mathbb{E}_2[x_1] \mid x_1 = \mathbb{E}_1$

evaluation contexts: $\mathbb{E} ::= [\cdot] \mid (\mathbb{E}\ e) \mid (\texttt{case}\ \mathbb{E}\ \texttt{of}\ alts) \mid (\texttt{seq}\ \mathbb{E}\ e)$

forcing contexts: $\mathbb{F} ::= \mathbb{E} \mid (\texttt{takeMVar}\ \mathbb{E}) \mid (\texttt{putMVar}\ \mathbb{E}\ e)$

# Program Equivalence

Process $P$ is **successful** if

$$P \text{ well-formed} \wedge P \equiv \nu\overrightarrow{x_i}(x \xLeftarrow{\text{main}} \mathtt{return}\ e \mid P')$$

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

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

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

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

**Contextual Equivalence**

$$P_1 \sim_c P_2 \text{ iff } \forall \mathbb{D} : (\mathbb{D}[P_1]{\downarrow} \iff \mathbb{D}[P_2]{\downarrow}) \wedge (\mathbb{D}[P_1]{\Downarrow} \iff \mathbb{D}[P_2]{\Downarrow})$$

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**

$\Downarrow, \downarrow, \leq_c, \sim_c$ 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

$P_1 \xrightarrow{sr} P_2 \xrightarrow{sr} \ldots$ has an infinite suffix $P_j \xrightarrow{sr} P_{j+1} \xrightarrow{sr} \ldots$
where a (reducible) thread is never reduced

# Context Lemma

A proof tool to show equivalences:

**Context Lemma for Expressions**

If $\forall \mathbb{D}[\mathbb{L}[\cdot^{\tau}]$-contexts:

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

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:

$$
\begin{array}{ll}
\text{(cpce)} & y \Leftarrow \mathbb{M}[\mathbb{F}[x]] \mid x = e \xrightarrow{src} y \Leftarrow \mathbb{M}[\mathbb{F}[e]] \mid x = e \\
\text{(nbeta)} & y \Leftarrow \mathbb{M}[\mathbb{F}[((\lambda x.e_1)\ e_2)]] \xrightarrow{src} y \Leftarrow \mathbb{M}[\mathbb{F}[e_1[e_2/x]]] \\
\text{(ncase)} & y \Leftarrow \mathbb{M}[\mathbb{F}[\mathtt{case}_T\ (c\ e_1\ \ldots\ e_n)\ \mathtt{of}\ \ldots((c\ y_1\ \ldots\ y_n) \to e)\ldots]] \\
 & \xrightarrow{src} y \Leftarrow \mathbb{M}[\mathbb{F}[e[e_1/y_1,\ldots,e_n/y_n]]]
\end{array}
$$

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

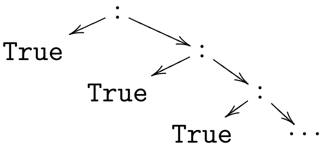
**Theorem**

$$P\downarrow \iff P\downarrow_{src} \text{ and } P\Downarrow \iff P\Downarrow_{src}$$

# Outline of the Proof

Translation $IT$ :: CHF $\to$ CHFI unfolds all bindings into infinite trees, e.g.

$$IT\begin{pmatrix}\texttt{letrec}\ xs = (\texttt{True} : xs) \\ \texttt{in}\ xs\end{pmatrix} =$$




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

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

**Correctness**

A transformation on processes $P_1 \to P_2$ is correct iff $P_1 \sim_c P_2$

A transformation on expressions $e_1 \to e_2$ 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:

$\nu x.\mathbb{D}[y \Leftarrow \mathbb{M}[\texttt{takeMVar}\ x] \mid x\, \mathbf{m}\, e] \to \nu x.\mathbb{D}[y \Leftarrow \mathbb{M}[\texttt{return}\ e] \mid x\, \mathbf{m} -]$
if no other `takeMVar` on $x$ is possible in any context

$\nu x.\mathbb{D}[y \Leftarrow \mathbb{M}[\texttt{putMVar}\ x\ e] \mid x\, \mathbf{m} -] \to \nu x.\mathbb{D}[y \Leftarrow \mathbb{M}[\texttt{return}\ ()] \mid x\, \mathbf{m}\, e]$
if no other `putMVar` on $x$ is possible in any context

# Further Transformations and Optimizations

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

## Results on other transformations

- General copying (gcp):

  (gcp) $\mathbb{C}[x] \mid x = e \rightarrow \mathbb{C}[e] \mid x = e$

- Garbage collection (gc):

  (gc) $\nu x_1, \ldots, x_n.(P \mid \texttt{Comp}(x_1) \mid \ldots \mid \texttt{Comp}(x_n)) \rightarrow P$

  where every $\texttt{Comp}(x_i)$ is
  - a binding $x_i = e_i$,
  - an MVar $x_i\,\mathbf{m}\,e_i$, or
  - an empty MVar $x_i\,\mathbf{m}\,-$

  and $x_i \notin FV(P)$.

# Monad Laws

**Theorem**

The monad laws

$$\begin{array}{llcl}
\text{(M1)} & \texttt{return}\ e_1 \ \texttt{>>=}\ e_2 & \sim_c & e_2\ e_1 \\
\text{(M2)} & e_1 \ \texttt{>>=}\ \lambda x.\texttt{return}\ x & \sim_c & e_1 \\
\text{(M3)} & e_1 \ \texttt{>>=}\ (\lambda x.(e_2\ x \ \texttt{>>=}\ e_3)) & \sim_c & (e_1 \ \texttt{>>=}\ e_2) \ \texttt{>>=}\ e_3
\end{array}$$

are correct.

$\Rightarrow$ use of do-notation is correct

```
do
 x1 ← future e1
 x2 ← future e2
 return (x1 + x2)
```

# Conclusion & Further Work

## Conclusion

- CHF models Concurrent Haskell with futures
- Contextual equivalence based on may- and should-convergence
- Call-by-need and 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`
  - ...