On Conservativity of pure Haskell in Concurrent Haskell

Manfred Schmidt-Schauß

joint work with David Sabel

Goethe-University, Frankfurt am Main, Germany

LogInf'11, Ilmenau

Motivation

- **Haskell** is a pure functional language, non-strict evaluation, polymorphic type system, lazy infinite data structures
- **Haskell**'s non-pure features use monadic programming; "monadic IO"
- **Concurrent Haskell** (Peyton Jones, Gordon, Finne 1996) extends Haskell by concurrency
  Further extensions: `unsafeInterleaveIO`, `unsafePerformIO`

- Proposal:
  Concurrent Haskell extended by **concurrent futures**:
  a more
  declarative programming style for concurrency

An Example Using Haskell and Futures

Do-notation of Haskell for monadic programming

```
do
   x1 <- future e1     Start computing e1 (thread 1)
   x2 <- future e2     Start computing e2 (thread 2)

   print (x1 + x2)     if x1, x2 are computed,
                       compute / print (x1 + x2)
```

An Example Using Haskell and Futures

Do-notation of Haskell for monadic programming

```
do
   x1 <- future e1     Start computing e1 (thread 1)
   x2 <- future e2     Start computing e2 (thread 2)

   print (x1 + x2)     if x1, x2 are computed,
                       compute / print (x1 + x2)
```

e1,e2:: (`IO` τ) may be

- `(let  x = di in seq x (return x))`, where d1, d2 are functional expressions ⇒ parallel evaluation
- a monadic expression; i.e., a sequence of side-effecting commands.

Issues

Is Concurrent Haskell with Futures “semantically sound”?

- Is partial evaluation correct?
- Do **monad laws** hold?

  Are monadic optimizations permitted?
- **Correctness** of **compiler optimizations** and **program transformations**

  Are **pure** transformations/optimizations also valid under concurrency?

Issues

Is Concurrent Haskell with Futures “semantically sound”?

- Is partial evaluation correct?

  Yes up to modifying storage (PPDP’11)

- Do **monad laws** hold?

  Are monadic optimizations permitted?

  Yes (PPDP’11) under a type restriction

- **Correctness** of **compiler optimizations** and **program transformations**

  Are **pure** transformations/optimizations also valid under concurrency?

  Yes (submitted)

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 ()`

Our Language Model

The (process) calculus CHF, inspired by (Peyton Jones, 2001);
A combination of

- A pure call-by-need extended lambda calculus
  with cyclic let (letrec), case, constructors, `seq`
- concurrent futures (variables whose value is determined by concurrent evaluation, final value not modifiable)
- MVars (modifiable storage, with synchronizing features)
- monadic programming
- A process calculus on top of the functional calculus
- monomorphic types

The Process Calculus CHF: Syntax

Processes

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

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

Functional Expressions & Monadic Expressions

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

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

The Process Calculus CHF: Syntax

Processes

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

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

Functional Expressions & Monadic Expressions

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

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

Process and Components

A process in prefix form:

$$\nu x_1, \ldots x_h . \underbrace{P_1 \mid \ldots \mid P_k}_{\substack{\text{futures} \\ x \Leftarrow e \\ (x \overset{\text{main}}{\Longleftarrow} e)}} \mid \underbrace{P_{k+1} \mid \ldots \mid P_m}_{\substack{\text{Bindings} \\ x = e}} \mid \underbrace{P_{m+1} \mid \ldots \mid P_n}_{\substack{\text{MVars} \\ x\, \mathbf{m}\, e \\ \text{or} \\ x\, \mathbf{m}\, -}}$$

The Process Calculus CHF: Typing

Syntax of (monomorphic) types

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

Type system:

- Usual monomorphic type system with recursive data constructors
- An extra condition holds for $\texttt{seq} :: \tau_1 \rightarrow \tau_2 \rightarrow \tau_2$

  τ_1 must not be an `IO`- or `MVar`-type

  τ_1: is a pure type at its top-level

Operational Semantics

Operational Semantics: Call-by-Need 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

where

$\equiv$ is a syntactic congruence similar as in the π-calculus

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

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

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

Semantics: Program Equivalence

Process P is **successful** if

$$P \text{ well-formed} \land P \equiv \nu \overrightarrow{x_i}(x \overset{\text{main}}{\Longleftarrow} \texttt{return } e \mid P')$$

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

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

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 as Semantics

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

Similar for expressions e_i of type τ: $e_1 \sim_{c,\tau} e_2$.

Results: Call-by-name Evaluation is Correct

Call-by-name Reduction

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

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

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

Theorem: call-by-name equivalent to call-by-need

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

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,lunit), (sr,nmvar), (sr,fork), (unIO) are correct
- (sr,tmvar) and (sr,pmvar) are in general not correct (as transformation (at compile time))
- Deterministic take and put are correct:

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

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

Further Transformations and Optimizations

Results on other transformations

- General copying (gcp) is correct:

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

- Garbage collection (gc) is correct:

  (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 **m** e_i, or
  - an empty MVar x_i **m** $-$

  and $x_i \notin FV(P)$.

Monad Laws

Theorem

The monad laws

$$
\begin{array}{llll}
\text{(M1)} & \mathtt{return}\ e_1 \ \gg\!= \ e_2 & \sim_c & e_2\ e_1 \\
\text{(M2)} & e_1 \ \gg\!= \ \lambda x.\mathtt{return}\ x & \sim_c & e_1 \\
\text{(M3)} & e_1 \ \gg\!= \ (\lambda x.(e_2\ x \ \gg\!= \ e_3)) & \sim_c & (e_1 \ \gg\!= \ e_2) \ \gg\!= \ e_3
\end{array}
$$

are correct in CHF.

The monad laws do not hold even in usual Haskell, if seq has unrestricted type! For example: (M1) does not hold:

```
Prelude> seq ((return True >>= undefined)::IO ()) True ⏎
True
Prelude> seq ((undefined True)::IO ()) True ⏎
*** Exception: Prelude.undefined
```

Conservativity: Embeddings and Infinite Calculi

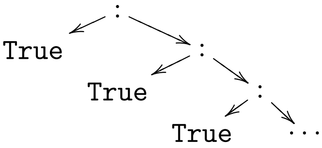
Goal: PF is conservatively embedded in CHF.

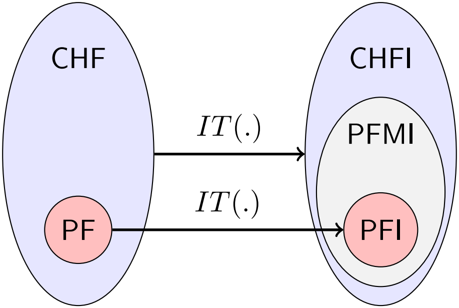
CHF	full calculus
PF	pure fragment of CHF: only functional expressions, letrec call-by-name, beta-, case-, seq-reductions.
CHFI	letrecs and top-bindings replaced by their infinite unfoldings
PFMI	pure fragment of CHFI: only infinite functional expressions, also monadic constructions call-by-name, beta-, case-, seq-reductions. no monadic reduction.
PFI	PF where letrec expressions are replaced by their infinite unfoldings.

Proof Idea: Infinite Expressions

Translation IT :: CHF → CHFI unfolds all bindings into infinite trees, e.g.

$$IT\left(\begin{array}{l}\texttt{letrec}\ xs = (\texttt{True} : xs)\\ \texttt{in}\ xs\end{array}\right) =$$



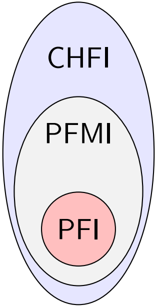


Steps of the Call-by-name = call-by-need proof

Some proof steps:

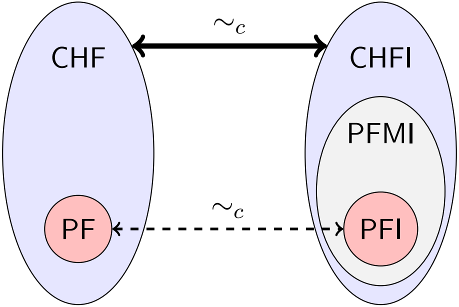
- Define call-by-name reduction on infinite trees
- Show $t{\downarrow}_{CHF} \iff IT(t){\downarrow}_{CHFI}$,
  and $t{\Downarrow}_{CHF} \iff IT(t){\Downarrow}_{CHFI}$,
  (convergence equivalence of tree reduction and call-by-need reduction)
- Show $t{\downarrow}_{src,CHF} \iff IT(t){\downarrow}_{CHFI}$,
  and $t{\Downarrow}_{src,CHF} \iff IT(t){\Downarrow}_{CHFI}$,
  (convergence equivalence of tree reduction and call-by-name reduction)

Steps of the Conservativity Proof



1. PFI: Equivalence of contextual ($\sim_c$) and applicative bisimulation ($\sim_b$) (by Howe's technique)
2. PFI $\xrightarrow{\iota}$ PFMI: embedding is conservative w.r.t $\sim_b$
3. PFMI $\xrightarrow{\iota}$ CHFI: embedding is conservative w.r.t. $\sim_c$: CHFI-Reduction is compatible with $\sim_{b,PFMI}$ lifted to CHFI.

Steps of the Conservativity Proof



1. CHF $\xrightarrow{IT(.)}$ CHFI is adequate and fully abstract w.r.t $\sim_c$
2. PF $\xrightarrow{IT(.)}$ PFI is adequate and fully abstract w.r.t $\sim_c$

Conservativity

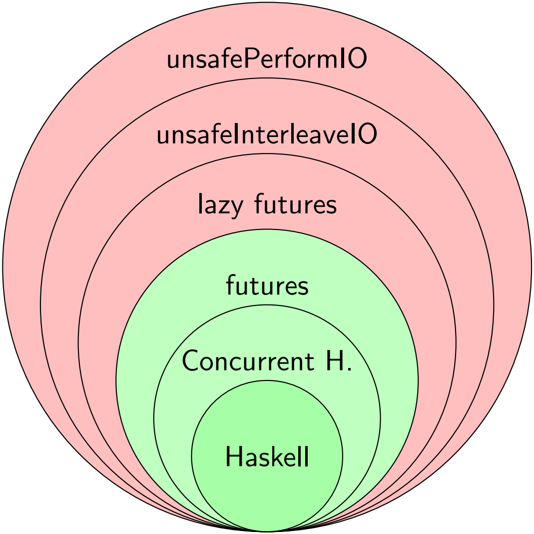
Theorem PF is conservatively embedded in CHF w.r.t. $\sim_c$

Consequences:

- program transformations valid in PF remain valid in CHF, like list-laws, for example: $(\mathtt{map}\ f).(\mathtt{map}\ g) = \mathtt{map}\ (f.g)$
- Compiler-optimizations valid in the deterministic pure language remain correct in the concurrent CHF.
- Presumably can be transferred to concurrent Haskell with futures.
  (Difference: polymorphic vs monomorphic typing)

Concurrent Haskell + Extensions

Conservativity of Haskell embedding in extensions: ok or wrong



Conclusion & Further Work

Conclusion

- 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 safe and available
- Conservativity of PF $\xrightarrow{\iota}$ CHF holds.
- Concurrent Haskell with futures is a safe extension of Haskell

Further Work

- Analyze further Concurrent Haskell extensions:
  - Exceptions
  - `killThread`
  - ...
- CHF + polymorphic types