Conservative Concurrency in Haskell

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

LICS’12, Dubrovnik, Croatia
Motivation – a View on Haskell

**Purely functional core language**
call-by-need lambda-calculus
with letrec, seq, data constructors

\[ \text{+ Monadic I/O} \approx \text{Haskell} \]

\[ \text{+ concurrent threads & MVars} \approx \text{Concurrent Haskell} \]

\[ \text{+ lazy I/O unsafePerformIO, unsafeInterleaveIO} \approx \text{Real implementations of Haskell} \]

- semantically **well-understood** & extensively **investigated**
- a lot of **correct** of program **transformations** & compiler **optimizations** are known
Issues

- Is the compiler still correct after extending the language?
- In short: Are these extensions safe?

**Safety = Conservativity**

An extension is **conservative** if it preserves program equalitites

\[ e_1 \sim_L e_2 \implies e_1 \sim_{L'} e_2 \]

\[ L \rightarrow \text{extension } L' \]

\[ \implies \text{correct program transformations of } L \text{ are also correct in } L' \]
Our Setting

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

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

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

- allow **implicit synchronisation by data dependency**

- Concurrent Haskell + `unsafeInterleaveIO` can encode CHF
  (CHF is a sublanguage of Concurrent Haskell + `unsafeInterleaveIO`)
The Process Calculus CHF

Processes

\[ P, P_i \in Proc ::= P_1 | P_2 | \nu x. P | x \leftarrow e | x = e | \leftarrow x \text{ future } x | x \text{ m e} | x \text{ m } - \]

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

Expressions

\[ e, e_i \in Expr_{CHF} ::= x | \lambda x. e | (e_1 e_2) | \text{seq } e_1 e_2 | e_1 \ldots e_{\text{ar}(c)} | \text{case } T e \text{ of } c_{T,i} \ x_1 \ldots x_{\text{ar}(c_{T,i})} \rightarrow e_i \ldots | \text{letrec } x_1 = e_1 \ldots x_n = e_n \text{ in } e | \text{return } e | e_1 >>= e_2 | \text{future } e | \text{takeMVar } e | \text{newMVar } e | \text{putMVar } e_1 e_2 \]

Types: standard monomorphic type system
Operational semantics: Small-step reduction relation $\xrightarrow{\text{CHF}}$

Process $P$ is successful if $P$ well-formed $\land P \equiv \nu x_i(x \leftarrow \text{return } e | P')$

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

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

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

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

Contextual Equivalence $\sim_{c, \text{CHF}}$

On processes:

$P_1 \sim_{c, \text{CHF}} P_2$ iff $\forall D: (D[P_1] \downarrow \iff D[P_2] \downarrow) \land (D[P_1] \downarrow \iff D[P_2] \downarrow)$

On expressions: $e_1, e_2 :: \tau$

$e_1 \sim_{c, \text{CHF}} e_2$ iff $\forall C: (C[e_1] \downarrow \iff C[e_2] \downarrow) \land (C[e_1] \downarrow \iff C[e_2] \downarrow)$
Conservativity

\[ PF = \text{Pure, deterministic sublanguage of CHF, no futures, no I/O} \]

\[ e, e_i \in Expr_{PF} ::= x \mid \lambda x. e \mid (e_1 e_2) \mid \text{seq } e_1 e_2 \mid c\ e_1 \ldots e_{\ar(c)} \]
\[ \mid \text{case}_T e \text{ of } \ldots (c_T,i\ x_1 \ldots x_{\ar(c_T,i)} \rightarrow e_i) \ldots \]
\[ \mid \text{letrec } x_1 = e_1 \ldots x_n = e_n \text{ in } e \]

**Main Theorem**

CHF extends PF conservatively

\[ \text{l.e., for all } e_1, e_2 :: \tau \in Expr_{PF}: e_1 \sim_{c,PF} e_2 \implies e_1 \sim_{c,CHF} e_2. \]

\[ \implies \text{ correct transformations of the pure core are still valid in CHF} \]
Outline of the Proof

\[ e_1 \sim_{c, CHF} e_2 \]

\[ e_1 \sim_{c, PF} e_2 \]
Outline of the Proof

Step 1: transport the problem to calculi with infinite trees:
- IT unfolds all bindings, $CHFI = IT(CHF)$ and $PFI = IT(PF)$

\[
ex = 1 : x \xrightarrow{IT} 1 \leadsto \ldots\]

David Sabel  Conservative Concurrency in Haskell
**Outline of the Proof**

Finite syntax

$e_1 \sim_{c, \text{CHF}} e_2$

Infinite trees

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

**Step 2:** define **bisimilarity** $\sim_{b, \text{PFI}}$ in $\text{PFI}$

- **Howe's method** shows $\sim_{b, \text{PFI}} = \sim_{c, \text{PFI}}$
Outline of the Proof

Step 3: add **monadic operators** (interpreted like constants) = $PFMI$

- bisimilarity is unchanged: $\sim_{b,PFI} = \sim_{b,PFMI}$
Outline of the Proof

Step 4: show $e_1 \sim_{b,PFMI} e_2 \implies e_1 \sim_{c,CHFI} e_2$
- syntactical proof by cases
Non-Conservativity Results

\[ \text{CHFL} = \text{CHF} + \text{lazy futures} \]

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

- \text{CHFL} is \textbf{not a conservative extension} of \texttt{PF}

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

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

- \( \text{CHF} + \text{unsafeInterleaveIO} \) is also \textbf{not a conservative extension} of \texttt{PF}
Conclusion

- CHF (and also Concurrent Haskell) are conservative extensions of the pure core language
- Result shown w.r.t. contextual equivalence based on may- and should-convergence
- Adding unsafeInterleaveIO (or even lazy futures) breaks conservativity

Future Work

- CHF with polymorphic typing
- Other primitives like exceptions