An Abstract Machine for Concurrent Haskell with Futures

David Sabel

Goethe-University, Frankfurt am Main, Germany

ATPS'12, Berlin, Germany

Motivation

- **Concurrent Haskell** (Peyton Jones, Gordon, Finne 1996) extends Haskell by concurrency
- The **process calculus CHF** (S.,Schmidt-Schauß 2011) models **Concurrent Haskell with Futures** operational semantics inspired by (Peyton Jones, 2001)
- Futures allow a **declarative programming style** for concurrency
  - Future = Variable whose value becomes available in the future
  - Our futures are concurrent, imperative, implicit
  - implicit synchronisation by data dependency

```
do
  x1 <- future e1
  x2 <- future e2
  some actions
  print(x1+x2)
```

$\longrightarrow$

$$x1 \Leftarrow e1 \;\mid\; x2 \Leftarrow e2 \;\mid\; \_ \overset{\text{main}}{\Longleftarrow} \texttt{do some actions print (x1+x2)}$$

Issues

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Operational semantics of CHF given in (S.,Schmidt-Schauß 2011)

- Appropriate for **mathematical reasoning** on **contextual equivalence** w.r.t. may- and should-convergence
- but its definition is **complex**
- **not** obvious how to implement

In this work:

- Design an **abstract machine** for CHF
  based on the machines of (Sestoft 1997) for lazy evaluation
- which can be implemented **easily** (prototype exists)
- show **correctness** of the abstract machine
  w.r.t. may- and should-convergence

The Process Calculus CHF: Syntax

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Processes

$P, P_i \in \mathit{Proc} ::= P_1 \mid P_2 \mid \nu x.P \mid x \Leftarrow e \mid x = e \mid x\,\mathbf{m}\,e \mid x\,\mathbf{m}\,-$

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

Expressions & Monadic Expressions

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

$me \in \mathit{MExpr} ::= \mathtt{return}\ e \mid e_1 \gg\!= e_2 \mid \mathtt{future}\ e$
$\mid \mathtt{takeMVar}\ e \mid \mathtt{newMVar}\ e \mid \mathtt{putMVar}\ e_1\ e_2$

Types

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

Standard monomorphic type system

Operational Semantics

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Operational Semantics: Reduction $P_1 \xrightarrow{CHF} P_2$

- Small-step reduction $\xrightarrow{CHF}$
- Rules are closed w.r.t. structural congruence and process contexts
- Reduction rules for monadic computation and functional evaluation

Some rules:

(fork) $x \Leftarrow \mathbb{M}[\texttt{future}\ e] \xrightarrow{CHF} \nu y.(x \Leftarrow \mathbb{M}[\texttt{return}\ y] \mid y \Leftarrow e),\ y$ fresh

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

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

Program Equivalence

Process P is **successful** if

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

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

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

Expression $e :: \texttt{IO}\ \tau$: $e{\downarrow}$ iff $x \overset{\text{main}}{\Longleftarrow} e{\downarrow}$

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

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

Expression $e :: \texttt{IO}\ \tau$: $e{\Downarrow}$ iff $x \overset{\text{main}}{\Longleftarrow} e{\Downarrow}$

Contextual Equivalence

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

Simplified Expressions and Processes

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Processes

$$P, P_i \in \mathit{Proc} \;::=\; P_1 \mid P_2 \;\mid\; \nu x.P \;\mid\; x \Leftarrow e \;\mid\; x = e \;\mid\; x\,\mathbf{m}\,e \;\mid\; x\,\mathbf{m}\,-$$

Expressions & Monadic Expressions

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

$$\begin{aligned} me \in \mathit{MExpr} ::= {} & \mathtt{return}\ e \mid e_1 \mathbin{\mathtt{>>=}} e_2 \mid \mathtt{future}\ e \\ & \mid \mathtt{takeMVar}\ e \mid \mathtt{newMVar}\ e \mid \mathtt{putMVar}\ e_1\ e_2 \end{aligned}$$

Simplified Expressions and Processes

Simplified Processes

$$P, P_i \in \mathit{Proc} \;::=\; P_1 \mid P_2 \;\mid\; \nu x.P \;\mid\; x \Leftarrow e \;\mid\; x = e \;\mid\; x\,\mathbf{m}\,y \;\mid\; x\,\mathbf{m}\,-$$

Simplified Expressions & Monadic Expressions

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

$$\begin{array}{rl} me \in \mathit{MExpr} ::= & \texttt{return}\ x \mid x_1 \gg\!= x_2 \mid \texttt{future}\ x \\ & \mid \texttt{takeMVar}\ x \mid \texttt{newMVar}\ x \mid \texttt{putMVar}\ x_1\ x_2 \end{array}$$

Constructing the Abstract Machine

Modular construction in three steps:

- evaluates pure expressions
- deterministic
- treats monadic operators like data constructors
- slight modification of Sestoft's mark 1

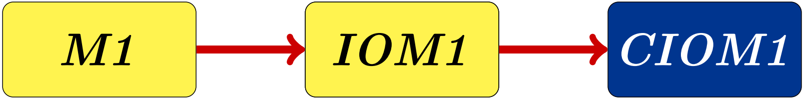
Constructing the Abstract Machine

Modular construction in three steps:

- adds storage (MVars)
- monadic operations are executed
- uses *M1* for purely functional subevaluations
- single-threaded

Constructing the Abstract Machine

Modular construction in three steps:



- Concurrent threads (futures)
- nondeterministic
- Globally shared bindings and MVars
- uses *IOM1* for thread-local evaluation

Machine *M1*

State

$$(\mathcal{H}, e, \mathcal{S})$$

- $\mathcal{H}$ is a **heap**: a set of shared bindings $x \mapsto e$
- e is the currently evaluated **expression**
- $\mathcal{S}$ is a **stack** (holding the evaluation context)
  entries: $\#_{\mathsf{app}}(x)$, $\#_{\mathsf{seq}}(x)$, $\#_{\mathsf{case}}(alts)$, $\#_{\mathsf{heap}}(x)$

Start state: For expression e: $(\emptyset, e, [])$

Final state: $(\mathcal{H}, v, [])$ where $v = \lambda z.e$, $v = (c \ldots)$,
or v a monadic expression

Machine $M1$: Transitions

Unwinding

(pushApp) $(\mathcal{H}, (e\ x), \mathcal{S}) \xrightarrow{M1} (\mathcal{H}, e, \#_{\text{app}}(x) : \mathcal{S})$

(pushSeq) $(\mathcal{H}, (\texttt{seq}\ e\ x), \mathcal{S}) \xrightarrow{M1} (\mathcal{H}, e, \#_{\text{seq}}(x) : \mathcal{S})$

(pushAlts) $(\mathcal{H}, \texttt{case}_T\ e\ \texttt{of}\ alts, \mathcal{S}) \xrightarrow{M1} (\mathcal{H}, e, \#_{\text{case}}(alts) : \mathcal{S})$

(mkBinds) $(\mathcal{H}, \texttt{letrec}\ \{x_i = e_i\}_{i=1}^n\ \texttt{in}\ e, \mathcal{S}) \xrightarrow{M1} (\mathcal{H} \dot\cup \bigcup_{i=1}^n \{x_i \mapsto e_i\}, e, \mathcal{S})$

(enter) $(\mathcal{H} \dot\cup \{y \mapsto e\}, y, \mathcal{S}) \xrightarrow{M1} (\mathcal{H}, e, \#_{\text{heap}}(y) : \mathcal{S})$

Evaluation

(takeApp) $(\mathcal{H}, \lambda x.e, \#_{\text{app}}(y) : \mathcal{S}) \xrightarrow{M1} (\mathcal{H}, e[y/x], \mathcal{S})$

(takeSeq) $(\mathcal{H}, v, \#_{\text{seq}}(y) : \mathcal{S}) \xrightarrow{M1} (\mathcal{H}, y, \mathcal{S})$, if $v = \lambda z.e$ or $v = c \ldots$

(branch) $(\mathcal{H}, (c\ \overrightarrow{x_i}), \#_{\text{case}}(\ldots (c\ \overrightarrow{y_i} \to e) \ldots) : \mathcal{S}) \xrightarrow{M1} (\mathcal{H}, e[x_i/y_i]_{i=1}^n, \mathcal{S})$

(update) $(\mathcal{H}, v, \#_{\text{heap}}(y) : \mathcal{S}) \xrightarrow{M1} (\mathcal{H} \dot\cup \{y \mapsto v\}, v, \mathcal{S})$
if $v = \lambda z.e$, $v = (c\ \ldots)$, $v = x$, or v a monadic operator

Example

$$
\begin{array}{rl}
 & (\emptyset, \texttt{letrec}\ x_1 = (\lambda y.y)\ w,\ x_2 = \texttt{takeMVar}\ x_1\ \texttt{in}\ ((\lambda z.z)\ x_2), [])\\
\xrightarrow{M1,\text{mkBinds}} & (\{x_1 \mapsto (\lambda y.y)\ w, x_2 \mapsto \texttt{takeMVar}\ x_1\}, (\lambda z.z)\ x_2, [])\\
\xrightarrow{M1,\text{pushApp}} & (\{x_1 \mapsto (\lambda y.y)\ w, x_2 \mapsto \texttt{takeMVar}\ x_1\}, \lambda z.z, [\#_{\text{app}}(x_2)])\\
\xrightarrow{M1,\text{takeApp}} & (\{x_1 \mapsto (\lambda y.y)\ w, x_2 \mapsto \texttt{takeMVar}\ x_1\}, x_2, [])\\
\xrightarrow{M1,\text{enter}} & (\{x_1 \mapsto (\lambda y.y)\ w\}, \texttt{takeMVar}\ x_1, [\#_{\text{heap}}(x_2)])\\
\xrightarrow{M1,\text{update}} & (\{x_1 \mapsto (\lambda y.y)\ w, x_2 \mapsto \texttt{takeMVar}\ x_1\}, \texttt{takeMVar}\ x_1, [])
\end{array}
$$

Machine *IOM1*

$M1$-state: $(\mathcal{H}, e, \mathcal{S})$

State

$$(\mathcal{H}, \mathcal{M}, e, \mathcal{S}, \mathcal{I})$$

- $\mathcal{H}$ is a heap
- e is the currently evaluated expression
- $\mathcal{S}$ is a stack (holding the evaluation context)
- $\mathcal{M}$ is a **set of MVars**: filled $x\,\mathbf{m}\,y$, empty $x\,\mathbf{m}\,-$
- $\mathcal{I}$ is an **IO-stack** (holding the monadic context)
  entries: $\#_{\mathsf{take}}, \#_{\mathsf{put}}(x), \#_{\gg=}(x)$

Start state: For expression $e :: \mathtt{IO}\ \tau$: $(\emptyset, \emptyset, e, [], [])$

Final state: $(\mathcal{H}, \mathcal{M}, \mathtt{return}\ x, [], [])$

Machine *IOM1*: Transitions

Functional Evaluation

(M1) $(\mathcal{H}, \mathcal{M}, e, \mathcal{S}, \mathcal{I}) \xrightarrow{IOM1} (\mathcal{H}', \mathcal{M}, e', \mathcal{S}', \mathcal{I})$
if $(\mathcal{H}, e, \mathcal{S}) \xrightarrow{M1} (\mathcal{H}', e', \mathcal{S}')$ on machine $M1$

Monadic Unwinding

(pushTake) $(\mathcal{H}, \mathcal{M}, \texttt{takeMVar}\ x, [], \mathcal{I}) \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M}, x, [], \#_{\mathsf{take}} : \mathcal{I})$

(pushPut) $(\mathcal{H}, \mathcal{M}, \texttt{putMVar}\ x\ y, [], \mathcal{I}) \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M}, x, [], \#_{\mathsf{put}}(y) : \mathcal{I})$

(pushBind) $(\mathcal{H}, \mathcal{M}, x \gg= y, [], \mathcal{I}) \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M}, x, [], \#_{\gg=}(y) : \mathcal{I})$

Monadic Computation

(newMVar) $(\mathcal{H}, \mathcal{M}, \texttt{newMVar}\ x, [], \mathcal{I}) \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M} \cup \{y\ \mathbf{m}\ x\}, \texttt{return}\ y, [], \mathcal{I})$
where y is a fresh variable

(takeMVar) $(\mathcal{H}, \mathcal{M} \cup \{x\ \mathbf{m}\ y\}, x, [], \#_{\mathsf{take}} : \mathcal{I}) \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M} \cup \{x\ \mathbf{m}\ -\}, \texttt{return}\ y, [], \mathcal{I})$

(putMVar) $(\mathcal{H}, \mathcal{M} \cup \{x\ \mathbf{m}\ -\}, x, [], \#_{\mathsf{put}}(y) : \mathcal{I}) \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M} \cup \{x\ \mathbf{m}\ y\}, \texttt{return}\ (), [], \mathcal{I})$

(lunit) $(\mathcal{H}, \mathcal{M}, \texttt{return}\ x, [], \#_{\gg=}(y) : \mathcal{I}) \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M}, (y\ x), [], \mathcal{I})$

Example

$$(\emptyset, \{w\,\mathbf{m}\,c\}, \texttt{letrec}\ x_1 = (\lambda y.y)\ w,\ x_2 = \texttt{takeMVar}\ x_1\ \texttt{in}\ ((\lambda z.z)\ x_2), [], [])$$

$$\xrightarrow{\mathit{IOM1},\mathrm{M1}} (\{x_1 \mapsto (\lambda y.y)\ w, x_2 \mapsto \texttt{takeMVar}\ x_1\}, \{w\,\mathbf{m}\,c\}, (\lambda z.z)\ x_2, [], [])$$

$$\xrightarrow{\mathit{IOM1},\mathrm{M1}} (\{x_1 \mapsto (\lambda y.y)\ w, x_2 \mapsto \texttt{takeMVar}\ x_1\}, \{w\,\mathbf{m}\,c\}, \lambda z.z, [\#_{\mathsf{app}}(x_2)], [])$$

$$\xrightarrow{\mathit{IOM1},\mathrm{M1}} (\{x_1 \mapsto (\lambda y.y)\ w, x_2 \mapsto \texttt{takeMVar}\ x_1\}, \{w\,\mathbf{m}\,c\}, x_2, [], [])$$

$$\xrightarrow{\mathit{IOM1},\mathrm{M1}} (\{x_1 \mapsto (\lambda y.y)\ w\}, \{w\,\mathbf{m}\,c\}, \texttt{takeMVar}\ x_1, [\#_{\mathsf{heap}}(x_2)], [])$$

$$\xrightarrow{\mathit{IOM1},\mathrm{M1}} (\{x_1 \mapsto (\lambda y.y)\ w, x_2 \mapsto \texttt{takeMVar}\ x_1\}, \{w\,\mathbf{m}\,c\}, \texttt{takeMVar}\ x_1, [], [])$$

$$\xrightarrow{\mathit{IOM1},\mathrm{pushTake}} (\{x_1 \mapsto (\lambda y.y)\ w, x_2 \mapsto \texttt{takeMVar}\ x_1\}, \{w\,\mathbf{m}\,c\}, x_1, [], [\#_{\mathsf{take}}])$$

$$\xrightarrow{\mathit{IOM1},\mathrm{M1}} (\{x_2 \mapsto \texttt{takeMVar}\ x_1\}, \{w\,\mathbf{m}\,c\}, (\lambda y.y)\ w, [\#_{\mathsf{heap}}(x_1)], [\#_{\mathsf{take}}])$$

$$\xrightarrow{\mathit{IOM1},\mathrm{M1}} (\{x_2 \mapsto \texttt{takeMVar}\ x_1\}, \{w\,\mathbf{m}\,c\}, \lambda y.y, [\#_{\mathsf{app}}(w), \#_{\mathsf{heap}}(x_1)], [\#_{\mathsf{take}}])$$

$$\xrightarrow{\mathit{IOM1},\mathrm{M1}} (\{x_2 \mapsto \texttt{takeMVar}\ x_1\}, \{w\,\mathbf{m}\,c\}, w, [\#_{\mathsf{heap}}(x_1)], [\#_{\mathsf{take}}])$$

$$\xrightarrow{\mathit{IOM1},\mathrm{M1}} (\{x_2 \mapsto \texttt{takeMVar}\ x_1, x_1 \mapsto w\}, \{w\,\mathbf{m}\,c\}, w, [], [\#_{\mathsf{take}}])$$

$$\xrightarrow{\mathit{IOM1},\mathrm{takeMVar}} (\{x_2 \mapsto \texttt{takeMVar}\ x_1, x_1 \mapsto w\}, \{w\,\mathbf{m}\,-\}, \texttt{return}\ c, [], [])$$

Machine *CIOM1*

IOM1-state: $(\mathcal{H}, \mathcal{M}, e, \mathcal{S}, \mathcal{I})$

State

$$(\mathcal{H}, \mathcal{M}, \mathcal{T})$$

- $\mathcal{H}$ is a heap
- $\mathcal{M}$ is a set of MVars
- $\mathcal{T}$ is a **set of threads**

Start state for expression $e :: \texttt{IO}\ \tau$: $\text{Init}(e) = (\emptyset, \emptyset, \{(x, e, [], [])^{\text{main}}\})$

Final state: Main-thread is of the form $(y, \texttt{return}\ x, [], [])^{\text{main}}$

Thread

$$(x, e, \mathcal{S}, \mathcal{I})$$

- x is a variable, the **name** of the future
- e is the currently evaluated expression
- $\mathcal{S}$ is a stack
- $\mathcal{I}$ is an IO-stack

Main-thread: $(x, e, \mathcal{S}, \mathcal{I})^{\text{main}}$.

Machine $\mathit{CIOM1}$: Transitions

Thread Evaluation

(IOM1) $(\mathcal{H}, \mathcal{M}, \mathcal{T} \dot\cup \{(x, e, \mathcal{S}, \mathcal{I})\}) \xrightarrow{\mathit{CIOM1}} (\mathcal{H}', \mathcal{M}', \mathcal{T} \dot\cup \{(x, e', \mathcal{S}', \mathcal{I}')\})$
if $(\mathcal{H}, \mathcal{M}, e, \mathcal{S}, \mathcal{I}) \xrightarrow{\mathit{IOM1}} (\mathcal{H}', \mathcal{M}', e', \mathcal{S}', \mathcal{I}')$ on machine $\mathit{IOM1}$.

Thread Creation and Finalization

(fork) $(\mathcal{H}, \mathcal{M}, \mathcal{T} \dot\cup \{(x, (\texttt{future}\ y), [], \mathcal{I})\})$
$\xrightarrow{\mathit{CIOM1}} (\mathcal{H}, \mathcal{M}, \mathcal{T} \dot\cup \{(x, (\texttt{return}\ z), [], \mathcal{I}), (z, y, [], [])\})$
where z is a fresh variable

(unIO) $(\mathcal{H}, \mathcal{M}, \mathcal{T} \dot\cup \{(x, (\texttt{return}\ y), [], [])\}) \xrightarrow{\mathit{CIOM1}} (\mathcal{H} \dot\cup \{x \mapsto y\}, \mathcal{M}, \mathcal{T})$
if thread named x is not the main-thread

Correctness

May- and should-convergence on $\mathit{CIOM1}$

State S:
- $S\downarrow_{\mathit{CIOM1}}$ iff $S \xrightarrow{\mathit{CIOM1},*} S' \wedge S'$ is a final state
- $S\Downarrow_{\mathit{CIOM1}}$ iff $\forall S' : S \xrightarrow{\mathit{CIOM1},*} S' \implies S'\downarrow_{\mathit{CIOM1}}$

Expression $e :: \mathtt{IO}\ \tau$
- $e\downarrow_{\mathit{CIOM1}}$ iff $\mathrm{Init}(\sigma(e))\downarrow_{\mathit{CIOM1}}$
- $e\Downarrow_{\mathit{CIOM1}}$ iff $\mathrm{Init}(\sigma(e))\Downarrow_{\mathit{CIOM1}}$

where σ translates usual expressions into simplified expressions

Theorem

For every expression $e :: \mathtt{IO}\ \tau$:

$$e\downarrow \iff e\downarrow_{\mathit{CIOM1}} \text{ and } e\Downarrow \iff e\Downarrow_{\mathit{CIOM1}}$$

Proof is not obvious, since transition on the machine is more restrictive than reduction in the process calculus

Conclusion & Further Work

Conclusion

- Sestoft's machine for lazy evaluation can be **modularly extended** to monadic I/O and concurrency
- *CIOM1* is a **correct** abstract machine for the process calculus CHF
- Correctness w.r.t. **may- and should-convergence**
- *CIOM1* is **easy to implement**, a prototype exists

Further Work

- **Optimize** *CIOM1* by using nameless representation and avoiding substitutions. Show correctness of the optimized machine.
- Investigate how to map *CIOM1*'s threads to **parallel architectures**