Der CHF-Kalkül als Modell für Concurrent Haskell

Prof. Dr. David Sabel

LFE Theoretische Informatik

Letzte Änderung der Folien: 21. Januar 2021

Übersicht

Der CHF-Kalkül

- CHF = Concurrent Haskell erweitert um (implizite) Futures
- Shared Memory Modell
- Speicher vorhanden durch MVars
- Futures = Nebenläufige Threads mit Rückgabewert
- Wie in Haskell: Seiteneffekte durch IO-Monade

Der CHF-Kalkül: Syntax

- Zweistufige Syntax: Oben Prozesskomponenten, unten (funktionale) Ausdrücke
- Prozesse $P \in \mathit{Proc}$:

$$
\begin{array}{rcll}
P, P_i \in \mathit{Proc} & ::= & P_1 \mid P_2 & \text{parallele Komposition} \\
& \mid & \nu x.P & \text{Namensbeschränkung} \\
& \mid & x \Leftarrow e & \text{nebenl. Thread (Future } x\text{)} \\
& \mid & x = e & \text{globale Bindung} \\
& \mid & x\,\mathbf{m}\,e & \text{gefüllte MVar} \\
& \mid & x\,\mathbf{m}\,- & \text{leere MVar}
\end{array}
$$

- Dabei: x Variable, e Ausdruck
- Ein Spezialthread möglich $x \overset{\text{main}}{\Longleftarrow} e$ der Main-Thread

Datenkonstruktoren

Es gibt Datenkonstruktoren $c_{T,i}$

- T ist der zugehörige Typkontruktor (z.B. Bool, List, etc.)
- Pro Typ gibt es Konstruktoren $c_{T,1}, \ldots, c_{T,|T|}$ (z.B. `True`, `False`)
- Konstruktoren haben eine feste Stelligkeit $ar(c_{T,i}) \in \mathbb{N}_0$
- Konstruktoren dürfen nur gesättigt auftreten: $(c_{T,i}\ e_1\ \ldots\ e_{ar(c_{T,i})})$
- Annahme: Es gibt Typ `()` mit nullstelligem Konstruktor `()`

Ausdrücke

Funktionale Ausdrücke

$$
\begin{array}{lll}
e, e_i \in \mathit{Exp} ::= & x & \text{Variable} \\
\mid & me & \text{monad. Ausdruck} \\
\mid & \lambda x.e & \text{Abstraktion} \\
\mid & (e_1\ e_2) & \text{Anwendung} \\
\mid & c\ e_1 \ldots e_{ar(c)} & \text{Konstruktoranw.} \\
\mid & \texttt{seq}\ e_1\ e_2 & \text{seq-Ausdruck} \\
\mid & \texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e & \texttt{letrec}\text{-Ausdruck} \\
\mid & \texttt{case}_T\ e\ \texttt{of}\ (c_{T,1}\ x_1 \ldots x_{ar(c_{T,1})} \to e_1) & \texttt{case}\text{-Ausdruck} \\
 & \ldots & \\
 & (c_{T,|T|}\ x_1 \ldots x_{ar(c_{T,|T|})} \to e_{|T|}) &
\end{array}
$$

Monadische Ausdrücke

$$
\begin{array}{l}
me \in \mathit{MExp} ::= \texttt{return}\, e \mid e_1 \gg\!= e_2 \mid \texttt{future}\, e \\
\quad \mid \texttt{takeMVar}\, e \mid \texttt{newMVar}\, e \mid \texttt{putMVar}\, e_1\, e_2
\end{array}
$$

Nebenbedingungen

$$\texttt{case}_T\ e\ \texttt{of}\ \overbrace{\underbrace{(c_{T,1}\ x_1 \ldots x_{ar(c_{T,1})}}_{\text{Pattern}} \to e_1)}^{\text{case-Alternative}}$$
$$\ldots$$
$$(c_{T,|T|}\ x_1 \ldots x_{ar(c_{T,|T|})} \to e_{|T|})$$

- Nur Variablen im Pattern erlaubt
- Variablen müssen paarweise verschieden sein.
- Pro $c_{T,i}$ genau eine Alternative

$$\texttt{letrec}\ \underbrace{x_1 = e_1}_{\texttt{letrec-Bindung}}, \ldots x_n = e_n\ \texttt{in}\ \underbrace{e}_{\texttt{in-Ausdruck}}$$

- Bindungsbereich von x_i: Alle e_i und e
- Alle x_i müssen paarweise verschieden sein.

Wohlgeformtheit, Strukturelle Kongruenz

- Eingeführte Variablen: der Name eines Threads, der Name einer MVar, die linke Seite einer Bindung
- Ein Prozess ist wohlgeformt gdw. alle eingeführten Variablen paarweise verschieden sind und es maximal einen Main-Thread gibt.

Strukturelle Kongruenz

$\equiv$ ist die kleineste Kongruenz auf Prozessen, die die Regeln erfüllt:

$$
\begin{aligned}
P_1 \mid P_2 &\equiv P_2 \mid P_1 \\
P_1 \mid (P_2 \mid P_3) &\equiv (P_1 \mid P_2) \mid P_3 \\
(\nu x.P_1) \mid P_2 &\equiv \nu x.(P_1 \mid P_2), \text{ falls } x \notin FV(P_2) \\
\nu x_1.\nu x_2.P &\equiv \nu x_2.\nu x_1.P \\
P_1 &\equiv P_2, \text{ falls } P_1 \text{ und } P_2 \ \alpha\text{-äquivalente } (P_1 =_\alpha P_2)
\end{aligned}
$$

Typisierung

- CHF ist monomorph typisiert
- $\tau, \tau_i \in \mathit{Typ} ::= \texttt{IO}\ \tau \mid (T\ \tau_1 \ldots \tau_{ar(T)}) \mid \texttt{MVar}\ \tau \mid \tau_1 \to \tau_2$
- Konstruktoren werden wie polymorph behandelt: `Cons::List Bool`, `Cons :: List (List Bool)` etc.
- Monadische Operatoren werden ebenfalls mit mehreren Typen verwendet.
- Annahme: Variablen x haben einen eingebauten Typ $\Gamma(x) \in \mathit{Typ}$
- $\Gamma \vdash P :: \texttt{wt}$ gdw. Prozess P ist wohlgetypt
- $\Gamma \vdash e :: \tau$ gdw. Ausdruck e ist wohlgetypt mit Typ τ.

Typisierung (2)

Einige Typisierungsregeln

$$\frac{\Gamma \vdash P_1 :: \mathtt{wt}, \Gamma \vdash P_2 :: \mathtt{wt}}{\Gamma \vdash P_1 \mid P_2 :: \mathtt{wt}} \qquad \frac{\Gamma \vdash P :: \mathtt{wt}}{\Gamma \vdash \nu x.P :: \mathtt{wt}} \qquad \frac{\Gamma \vdash x :: \tau, \Gamma \vdash e :: \mathtt{IO}\ \tau}{\Gamma \vdash x \Leftarrow e :: \mathtt{wt}}$$

$$\frac{\Gamma \vdash x :: \tau, \Gamma \vdash e :: \tau}{\Gamma \vdash x = e :: \mathtt{wt}} \qquad \frac{\Gamma \vdash x :: \mathtt{MVar}\ \tau, \Gamma \vdash e :: \tau}{\Gamma \vdash x\, \mathbf{m}\, e :: \mathtt{wt}} \qquad \frac{\Gamma \vdash x :: \mathtt{MVar}\ \tau}{\Gamma \vdash x\, \mathbf{m} - :: \mathtt{wt}}$$

$$\frac{\Gamma \vdash e :: \tau}{\Gamma \vdash \mathtt{return}\ e :: \mathtt{IO}\ \tau} \qquad \frac{\Gamma \vdash e_1 :: \mathtt{IO}\ \tau_1, \Gamma \vdash e_2 :: \tau_1 \to \mathtt{IO}\ \tau_2}{\Gamma \vdash e_1 \gg\!= e_2 :: \mathtt{IO}\ \tau_2}$$

$$\frac{\Gamma \vdash e :: \mathtt{IO}\ \tau}{\Gamma \vdash \mathtt{future}\ e :: \mathtt{IO}\ \tau} \qquad \frac{\Gamma \vdash e :: \mathtt{MVar}\ \tau}{\Gamma \vdash \mathtt{takeMVar}\ e :: \mathtt{IO}\ \tau}$$

$$\frac{\Gamma \vdash e_1 :: \mathtt{MVar}\ \tau, \Gamma \vdash e_2 :: \tau}{\Gamma \vdash \mathtt{putMVar}\ e_1\ e_2 :: \mathtt{IO}\ ()} \qquad \frac{\Gamma \vdash e :: \tau}{\Gamma \vdash \mathtt{newMVar}\ e :: \mathtt{IO}\ (\mathtt{MVar}\ \tau)}$$

Typisierung: Beispiele

$$\frac{\dfrac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau}, \dfrac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau}}{\Gamma \vdash (\lambda x.x) :: \tau \to \tau}$$

Typisierung: Beispiele

$$\frac{\dfrac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau}, \dfrac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau}}{\Gamma \vdash (\lambda x.x) :: \tau \to \tau}$$

$$\frac{\dfrac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau}, \dfrac{\dfrac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau}, \dfrac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau}}{(x\ x) ::?}}{\lambda x.(x\ x) :: \tau \to ?}$$

$$\frac{\Gamma \vdash e_1 :: \tau_1 \to \tau_2,\ \Gamma \vdash e_2 :: \tau_1}{\Gamma \vdash (e_1\ e_2) :: \tau_2}$$

Typisierung: Beispiele

$$\frac{\dfrac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau}, \dfrac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau}}{\Gamma \vdash (\lambda x.x) :: \tau \to \tau}$$

$$\frac{\dfrac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau}, \dfrac{\dfrac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau}, \dfrac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau}}{(x\ x) ::?}}{\lambda x.(x\ x) :: \tau \to ?} \qquad \frac{\Gamma \vdash e_1 :: \tau_1 \to \tau_2,\ \Gamma \vdash e_2 :: \tau_1}{\Gamma \vdash (e_1\ e_2) :: \tau_2}$$

$$id = \lambda x.x \mid y = id\ \texttt{True} \mid z = id\ \texttt{Nil}$$

Typisierung: Beispiele

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau}, \frac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau}$$
$$\overline{\Gamma \vdash (\lambda x.x) :: \tau \to \tau}$$

$$\frac{\dfrac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau}, \dfrac{\dfrac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau}, \dfrac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau}}{(x\ x) ::?}}{\lambda x.(x\ x) :: \tau \to ?} \qquad \frac{\Gamma \vdash e_1 :: \tau_1 \to \tau_2,\ \Gamma \vdash e_2 :: \tau_1}{\Gamma \vdash (e_1\ e_2) :: \tau_2}$$

$$id = \underbrace{\lambda x.x}_{\tau \to \tau} \mid y = \underbrace{id}_{\texttt{Bool} \to \texttt{Bool}}\ \texttt{True} \mid z = \underbrace{id}_{\texttt{List}\ \tau \to \texttt{List}\ \tau}\ \texttt{Nil}$$

Typisierung: Beispiele

$$\frac{\dfrac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau}, \dfrac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau}}{\Gamma \vdash (\lambda x.x) :: \tau \to \tau}$$

$$\frac{\dfrac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau}, \dfrac{\dfrac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau}, \dfrac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau}}{(x\ x) ::?}}{\lambda x.(x\ x) :: \tau \to ?} \qquad \frac{\Gamma \vdash e_1 :: \tau_1 \to \tau_2,\ \Gamma \vdash e_2 :: \tau_1}{\Gamma \vdash (e_1\ e_2) :: \tau_2}$$

$$id = \underbrace{\lambda x.x}_{\texttt{Bool}\to\texttt{Bool}} \mid id' = \underbrace{\lambda x'.x'}_{\texttt{List}\ \tau\to\texttt{List}\ \tau} \mid y = \underbrace{id}_{\texttt{Bool}\to\texttt{Bool}}\ \texttt{True} \mid z = \underbrace{id'}_{\texttt{List}\ \tau\to\texttt{List}\ \tau}\ \texttt{Nil}$$

Operationale Semantik: Kontexte

Prozesskontexte:

$$\mathbb{D} \in \mathit{PC} ::= [\cdot] \mid \mathbb{D} \mid P \mid P \mid \mathbb{D} \mid \nu x.\mathbb{D}$$

Monadische Kontexte:

$$\mathbb{M} \in \mathit{MC} ::= [\cdot] \mid \mathbb{M} \gg\!= e$$

Evaluations- und Forcingkontexte

$$\mathbb{E} \in \mathit{EC} ::= [\cdot] \mid (\mathbb{E}\ e) \mid (\texttt{case}\ \mathbb{E}\ \texttt{of}\ alts) \mid (\texttt{seq}\ \mathbb{E}\ e)$$
$$\mathbb{F} \in \mathit{FC} ::= \mathbb{E} \mid (\texttt{takeMVar}\ \mathbb{E}) \mid (\texttt{putMVar}\ \mathbb{E}\ e)$$

Operationale Semantik: *LC*-Kontexte

$$\begin{aligned}\mathbb{L} \in \mathit{LC} ::= {} & x \Leftarrow \mathbb{M}[\mathbb{F}] \\ & \mid x \Leftarrow \mathbb{M}[\mathbb{F}[x_n]] \mid x_n = \mathbb{E}_n[x_{n-1}] \mid \ldots \mid x_2 = \mathbb{E}_2[x_1] \mid x_1 = \mathbb{E}_1\end{aligned}$$

wobei $\mathbb{E}_2, \ldots \mathbb{E}_n$ nicht der leere Kontext sind.

$$\begin{aligned}\widehat{\mathbb{L}} \in \widehat{\mathit{LC}} ::= {} & x \Leftarrow \mathbb{M}[\mathbb{F}] \\ & \mid x \Leftarrow \mathbb{M}[\mathbb{F}[x_n]] \mid x_n = \mathbb{E}_n[x_{n-1}] \mid \ldots \mid x_2 = \mathbb{E}_2[x_1] \mid x_1 = \mathbb{E}_1\end{aligned}$$

wobei $\mathbb{E}_1, \mathbb{E}_2, \ldots \mathbb{E}_n$ nicht der leere Kontext sind.

Standardreduktion $\xrightarrow{CHF}$

Monadische Berechnungen:

(CHF,lunit) $y \Leftarrow \mathbb{M}[\texttt{return}\ e_1 \gg\!= e_2] \xrightarrow{CHF} y \Leftarrow \mathbb{M}[e_2\ e_1]$

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

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

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

(CHF,fork) $y \Leftarrow \mathbb{M}[\texttt{future}\ e] \xrightarrow{CHF} \nu z.(y \Leftarrow \mathbb{M}[\texttt{return}\ z] \mid z \Leftarrow e)$
wobei z ein neuer Name ist und der erzeugte Thread kein Main-Thread ist

(CHF,unIO) $y \Leftarrow \texttt{return}\ e \xrightarrow{CHF} y = e$,
wenn Thread y kein Main-Thread ist

Standardreduktion $\xrightarrow{CHF}$ (2)

Funktionale Auswertung:

(CHF,cp) $\widehat{\mathbb{L}}[x] \mid x = v \xrightarrow{CHF} \widehat{\mathbb{L}}[v] \mid x = v$,
falls v eine Abstraktion oder eine Variable ist

(CHF,cpcx) $\widehat{\mathbb{L}}[x] \mid x = c\ e_1 \ldots e_n$
$\xrightarrow{CHF} \nu y_1, \ldots y_n.(\widehat{\mathbb{L}}[c\ y_1 \ldots y_n] \mid x = c\ y_1 \ldots y_n \mid y_1 = e_1 \mid \ldots \mid y_n = e_n)$
falls c ein Konstruktor, oder ein monadischer Operator ist

$(CHF,\text{mkbinds})$ $\mathbb{L}[\texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e]$
$\xrightarrow{CHF} \nu x_1, \ldots, x_n.(\mathbb{L}[e] \mid x_1 = e_1 \mid \ldots \mid x_n = e_n)$

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

(CHF,case) $\mathbb{L}[\texttt{case}_T\ (c\ e_1\ \ldots\ e_n)\ \texttt{of}\ \ldots (c\ y_1\ \ldots\ y_n \to e)\ldots]$
$\xrightarrow{CHF} \nu y_1, \ldots, y_n.(\mathbb{L}[e] \mid y_1 = e_1 \mid \ldots \mid y_n = e_n])$

(CHF,seq) $\mathbb{L}[(\texttt{seq}\ v\ e)] \xrightarrow{CHF} \mathbb{L}[e]$, wenn v ein funktionaler Wert ist

Standardreduktion: Abschluss

$$\frac{P_1 \equiv \mathbb{D}[P_1'],\ P_2 \equiv \mathbb{D}[P_2'] \text{ und } P_1' \xrightarrow{CHF} P_2'}{P_1 \xrightarrow{CHF} P_2}$$

Erfolgreiche Prozesse

Definition

Ein wohlgeformter Prozess P ist genau dann erfolgreich, wenn er von der Form $\nu x_1, \ldots, x_n.x \overset{\text{main}}{\Longleftarrow} \texttt{return}\ e \mid P'$ ist.

- $\xrightarrow{CHF,+}$ bezeichnet die transitive Hülle von $\xrightarrow{CHF}$ (eine oder mehr Reduktionen)
- $\xrightarrow{CHF,*}$ bezeichnet die reflexiv-transitive Hülle (null oder mehr Reduktionen)

May- und Should-Konvergenz

Definition

May-Konvergenz:

$P{\downarrow_{CHF}}$ g.d.w. $\exists P' : P \xrightarrow{CHF,*} P'$ und P' ist erfolgreich

Should-Konvergenz:

$P{\Downarrow_{CHF}}$ g.d.w. $\forall P' : P \xrightarrow{CHF,*} P' \implies P'{\downarrow_{CHF}}$

Must-Konvergenz:

P should-konvergent

und es gibt keine unendlich lange Reduktion von P aus

Should-Konvergenz $\neq$ Must-Konvergenz

Beispiel

$$
\begin{aligned}
& x \overset{\text{main}}{\Longleftarrow} \texttt{future}\ (loopPut\ \texttt{True}) \gg\!= \lambda\_.\texttt{future}\ (loopPut\ \texttt{False}) \gg\!= \lambda\_.loop \\
& \mid loop = \texttt{takeMVar}\ x \ \gg\!= \\
& \qquad \lambda\_.\texttt{takeMVar}\ x \ \gg\!= \\
& \qquad\quad \lambda w.\texttt{case}_{Bool}\ w\ \texttt{of}\ (\texttt{True} \to \texttt{return True})\ (\texttt{False} \to loop) \\
& \mid loopPut = \lambda z.\texttt{putMVar}\ x\ z \ \gg\!= \ \lambda\_.loopPut\ z \\
& \mid x\ \mathbf{m} -
\end{aligned}
$$

Should-Konvergenz $\neq$ Must-Konvergenz

Beispiel mit syntaktischem Zucker

$$
\begin{aligned}
x \overset{\text{main}}{\Longleftarrow} & \texttt{do future } (loopPut\ \texttt{True}) \\
& \texttt{future } (loopPut\ \texttt{False}) \\
& loop \\
\mid loop = & \texttt{do takeMVar } x \\
& \quad w \leftarrow \texttt{takeMVar } x \\
& \quad \texttt{if } w \\
& \quad \texttt{ then return True} \\
& \quad \texttt{ else } loop \\
\mid loopPut = & \lambda z.\texttt{do putMVar } x\ z \\
& \qquad\qquad loopPut\ z \\
\mid x\,\mathbf{m} - &
\end{aligned}
$$

Should-Konvergenz $\neq$ Must-Konvergenz

Beispiel mit syntaktischem Zucker

$x \overset{\text{main}}{\Longleftarrow}$ `do future` $(loopPut$ `True`$)$ Thread, der wiederh. `True` in MVar x schreibt
`future` $(loopPut$ `False`$)$ Thread, der wiederh. `False` in MVar x schreibt
$loop$

$\mid loop =$ `do takeMVar` x 1x Lesen
$w \leftarrow$ `takeMVar` x 1x Lesen
`if` w wenn True, dann terminiere, sonst von vorne
`then return True`
`else` $loop$

$\mid loopPut = \lambda z.$`do putMVar` $x\ z$
$loopPut\ z$

$\mid x\,\mathbf{m} -$

Should-Konvergenz $\neq$ Must-Konvergenz

Beispiel mit syntaktischem Zucker

$$x \overset{\text{main}}{\Longleftarrow} \texttt{do future } (loopPut\ \texttt{True}) \text{ Thread, der wiederh. True in MVar } x \text{ schreibt}$$
$$\texttt{future } (loopPut\ \texttt{False}) \text{ Thread, der wiederh. False in MVar } x \text{ schreibt}$$
$$loop$$

$$\mid loop = \texttt{do takeMVar } x \text{ 1x Lesen}$$
$$w \leftarrow \texttt{takeMVar } x \text{ 1x Lesen}$$
$$\texttt{if } w \text{ wenn True, dann terminiere, sonst von vorne}$$
$$\texttt{then return True}$$
$$\texttt{else } loop$$

$$\mid loopPut = \lambda z.\texttt{do putMVar } x\ z$$
$$loopPut\ z$$

$$\mid x\ \mathbf{m} -$$

Prozess ist should-konvergent, aber nicht must-konvergent

May- und Must-Divergenz

Must-Divergenz: $P\Uparrow_{CHF}$ gdw. $\neg P\downarrow_{CHF}$

May-Divergenz: $P\uparrow_{CHF}$ gdw. $\neg P\Downarrow_{CHF}$

Satz

Für alle Prozesse P gilt: $P\uparrow_{CHF} \iff \exists P' : P \xrightarrow{CHF,*} P' \wedge P'\Uparrow_{CHF}$

Kontextuelle Gleichheit

Kontextuelle Approximation: $P_1 \leq_{CHF} P_2$ gdw.$P_1 \lesssim_{CHF} P_2$ und $P_1 \lesssim_{\Downarrow CHF} P_2$, wobei

$$P_1 \lesssim_{\downarrow CHF} P_2 \quad \text{gdw. } \forall \mathbb{D} \in \mathit{PC} : \mathbb{D}[P_1]{\downarrow_{CHF}} \implies \mathbb{D}[P_2]{\downarrow_{CHF}}$$
$$P_1 \lesssim_{\Downarrow CHF} P_2 \quad \text{gdw. } \forall \mathbb{D} \in \mathit{PC} : \mathbb{D}[P_1]{\Downarrow_{CHF}} \implies \mathbb{D}[P_2]{\Downarrow_{CHF}}$$

Kontextuelle Gleichheit $\sim_{CHF}$ auf Prozessen:

$$P_1 \sim_{CHF} P_2 \text{ gdw. } P_1 \leq_{CHF} P_2 \text{ und } P_2 \leq_{CHF} P_1$$

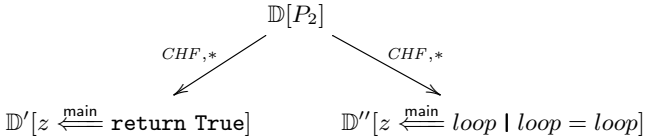
May-Konvergenz alleine reicht nicht

$$P_1 := \nu z.(z \overset{\text{main}}{\Longleftarrow} \texttt{return True})$$
$$P_2 := \nu x, z, y_1, y_2, loop.$$
$$(z \overset{\text{main}}{\Longleftarrow} \texttt{takeMVar}\ x \gg\!= \lambda w.\texttt{case}_{Bool}\ w\ (\texttt{True} \to \texttt{return True})\ (\texttt{False} \to loop)$$
$$\mid loop = loop \mid y_1 \Leftarrow \texttt{putMVar}\ x\ \texttt{False} \mid y_2 \Leftarrow \texttt{putMVar}\ x\ \texttt{True} \mid x\,\mathbf{m}\,-)$$

May-Konvergenz alleine reicht nicht

$$P_1 := \nu z.(z \overset{\text{main}}{\Longleftarrow} \texttt{return True})$$
$$P_2 := \nu x, z, y_1, y_2, loop.$$
$$(z \overset{\text{main}}{\Longleftarrow} \texttt{takeMVar}\ x \gg\!= \lambda w.\texttt{case}_{Bool}\ w\ (\texttt{True} \to \texttt{return True})\ (\texttt{False} \to loop)$$
$$\mid loop = loop \mid y_1 \Leftarrow \texttt{putMVar}\ x\ \texttt{False} \mid y_2 \Leftarrow \texttt{putMVar}\ x\ \texttt{True} \mid x\,\mathbf{m} -)$$

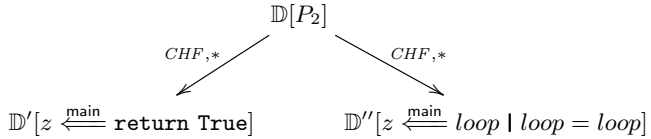
- $\mathbb{D}[P_1]$ ist für alle $\mathbb{D}$ direkt erfolgreich
- Für $\mathbb{D}[P_2]$ kann man zeigen:



May-Konvergenz alleine reicht nicht

$$P_1 := \nu z.(z \overset{\text{main}}{\Longleftarrow} \texttt{return True})$$
$$P_2 := \nu x, z, y_1, y_2, loop.$$
$$(z \overset{\text{main}}{\Longleftarrow} \texttt{takeMVar}\ x \gg\!= \lambda w.\texttt{case}_{Bool}\ w\ (\texttt{True} \to \texttt{return True})\ (\texttt{False} \to loop)$$
$$\mid loop = loop \mid y_1 \Leftarrow \texttt{putMVar}\ x\ \texttt{False} \mid y_2 \Leftarrow \texttt{putMVar}\ x\ \texttt{True} \mid x\,\mathbf{m} -)$$

- $\mathbb{D}[P_1]$ ist für alle $\mathbb{D}$ direkt erfolgreich
- Für $\mathbb{D}[P_2]$ kann man zeigen:



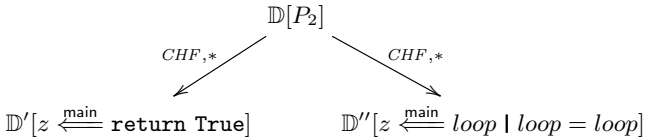
Daher gilt:

- Für alle $\mathbb{D} \in PC : \mathbb{D}[P_1]{\downarrow_{CHF}} \iff \mathbb{D}[P_2]{\downarrow_{CHF}}$
- Aber: $P_1{\Downarrow_{CHF}}$ während $P_2{\uparrow_{CHF}}$

Should-Konvergenz alleine reicht nicht

$$P_1 := \nu z, loop.(z \overset{\text{main}}{\Longleftarrow} loop) \mid loop = loop)$$
$$P_2 := \nu x, z, y_1, y_2, loop.$$
$$(z \overset{\text{main}}{\Longleftarrow} \texttt{takeMVar}\ x \gg\!= \lambda w.\texttt{case}_{Bool}\ w\ (\texttt{True} \to \texttt{return True})\ (\texttt{False} \to loop)$$
$$\mid loop = loop \mid y_1 \Leftarrow \texttt{putMVar}\ x\ \texttt{False} \mid y_2 \Leftarrow \texttt{putMVar}\ x\ \texttt{True} \mid x\,\mathbf{m} -)$$

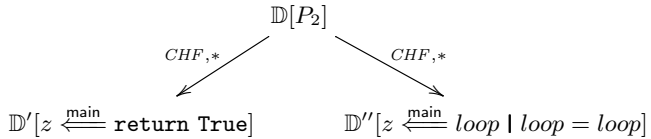
- $\mathbb{D}[P_1]$ ist für alle $\mathbb{D}$ must-divergent
- Für $\mathbb{D}[P_2]$ kann man zeigen:



Should-Konvergenz alleine reicht nicht

$$P_1 := \nu z, loop.(z \overset{\text{main}}{\Longleftarrow} loop) \mid loop = loop)$$
$$P_2 := \nu x, z, y_1, y_2, loop.$$
$$(z \overset{\text{main}}{\Longleftarrow} \texttt{takeMVar}\ x \gg\!= \lambda w.\texttt{case}_{Bool}\ w\ (\texttt{True} \to \texttt{return True})\ (\texttt{False} \to loop)$$
$$\mid loop = loop \mid y_1 \Leftarrow \texttt{putMVar}\ x\ \texttt{False} \mid y_2 \Leftarrow \texttt{putMVar}\ x\ \texttt{True} \mid x\,\mathbf{m} -)$$

- $\mathbb{D}[P_1]$ ist für alle $\mathbb{D}$ must-divergent
- Für $\mathbb{D}[P_2]$ kann man zeigen:



Daher gilt:

- Für alle $\mathbb{D} \in PC : \mathbb{D}[P_1]{\Downarrow_{CHF}} \iff \mathbb{D}[P_2]{\Downarrow_{CHF}}$
- Aber: $P_1{\Uparrow_{CHF}}$ während $P_2{\downarrow_{CHF}}$

Fairness

Die Reduktion $\xrightarrow{CHF}$ beachtet keine Fairness.

Beispiel:

$$
\begin{array}{ll}
 & x \xLeftarrow{\text{main}} \texttt{takeMVar}\ z \mid z\, \mathbf{m}\, \texttt{True} \mid y \Leftarrow loop \mid loop = loop \\
\xrightarrow{CHF,cp} & x \xLeftarrow{\text{main}} \texttt{takeMVar}\ z \mid z\, \mathbf{m}\, \texttt{True} \mid y \Leftarrow loop \mid loop = loop \\
\xrightarrow{CHF,cp} & x \xLeftarrow{\text{main}} \texttt{takeMVar}\ z \mid z\, \mathbf{m}\, \texttt{True} \mid y \Leftarrow loop \mid loop = loop \\
\xrightarrow{CHF,cp} & x \xLeftarrow{\text{main}} \texttt{takeMVar}\ z \mid z\, \mathbf{m}\, \texttt{True} \mid y \Leftarrow loop \mid loop = loop \\
\xrightarrow{CHF,cp} & \ldots
\end{array}
$$

Fairness (2)

Ausführbarer Thread: Für einen Prozess $P \equiv \mathbb{D}[x \Leftarrow e]$ ist der Thread x ausführbar, wenn es eine Reduktion $P \xrightarrow{CHF} P'$ gibt, die entweder innerhalb des Ausdrucks e reduziert, oder eine Reduktion ausführt, an der Thread x beteiligt ist (z.B. eine MVar liest oder schreibt, oder in e wird der Wert einer Bindung kopiert).

Definition

Für einen Prozess P ist die Reduktionsfolge $S = P \xrightarrow{CHF} P_1 \xrightarrow{CHF} P_2 \ldots$ unfair, wenn S einen unendlich langen Suffix S' hat, in dem ein Thread x unendlich oft ausführbar ist, aber niemals reduziert wird. Eine Reduktionsfolge ist fair, wenn sie nicht unfair ist.

Fairness (3)

- Faire May-Konvergenz $P{\downarrow_{CHF,f}}$ und Faire Should-Konvergenz $P{\Downarrow_{CHF,f}}$
- Wie May-Konvergenz und Should-Konvergenz, aber nur faire Reduktionsfolgen sind erlaubt.

Satz

$$\downarrow_{CHF} \;=\; \downarrow_{CHF,f} \quad \text{und} \quad \Downarrow_{CHF} \;=\; \Downarrow_{CHF,f}$$

Beweis: Siehe Skript

Vorteil: Wir brauchen uns um die Fairness nicht zu kümmern

Theorem

Kontextuelle Äquivalenz in CHF bleibt unverändert, wenn unfaire Reduktionssequenzen verboten sind.

Resultat gilt nicht für die Must-Konvergenz!

Programmtransformationen

- Eine Programmtransformation T ist eine binäre Relation auf Prozessen
- T ist korrekt, gdw. für alle $P, P' : P \xrightarrow{T} P' \implies P \sim_{CHF} P'$
- Korrektheit widerlegen ist eher einfach, da ein Kontext als Gegenbeispiel genügt
- Korrektheit beweisen ist eher schwierig, da alle Kontexte betrachtet werden müssen

Satz

Der Nachweis und die Widerlegung der Korrektheit einer Programmtransformation ist unentscheidbar.

Beweis: Reduktion des Halteproblems: Die Aussage
$(P \not\sim_{CHF} x \xleftarrow{\text{main}} \texttt{letrec}\ y = y\ \texttt{in}\ y)$ entspricht dem Halteproblem

Ungleichheit - Beispiel

- $P_1 := x \Leftarrow \texttt{return True}$
- $P_2 := x \Leftarrow \texttt{return False}$
- $\mathbb{D} := [\cdot] \mid y \overset{\text{main}}{\Longleftarrow} \texttt{case}_{Bool}\ x$
  $(\texttt{True} \rightarrow \texttt{returnTrue})$
  $(\texttt{False} \rightarrow \texttt{letrec}\ w = w\ \texttt{in}\ w)$
- $\mathbb{D}[P_1]{\downarrow_{CHF}}$ aber $\mathbb{D}[P_2]{\Downarrow_{CHF}}$.
- Daher $P_1 \not\sim_{CHF} P_2$

Einige korrekte Programmtransformationen

Satz

Die Reduktionen $(CHF, lunit)$, $(CHF, nmvar)$, $(CHF, fork)$, $(CHF, unIO)$, $(CHF, mkbinds)$ sind korrekte Programmtransformationen.

Beweis:

- Sei $P_1' \xrightarrow{a} P_2'$ wobei a wie im Satz und $P_1 \equiv \mathbb{D}[P_1']$ und $P_2 \equiv \mathbb{D}[P_2']$
- Wir müssen vier Implikationen zeigen:

(1) $P_1{\downarrow_{CHF}} \Longrightarrow P_2{\downarrow_{CHF}}$ (2) $P_2{\downarrow_{CHF}} \Longrightarrow P_1{\downarrow_{CHF}}$

(3) $P_1{\Downarrow_{CHF}} \Longrightarrow P_2{\Downarrow_{CHF}}$ (4) $P_2{\Downarrow_{CHF}} \Longrightarrow P_1{\Downarrow_{CHF}}$

Einige korrekte Programmtransformationen

Vorüberlegungen:

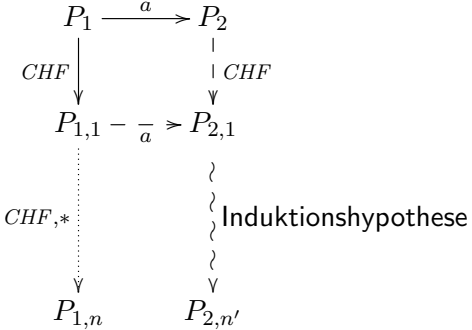
- Wenn $P_1 \xrightarrow{a} P_2$ und P_1 ist erfolgreich, dann muss auch P_2 erfolgreich sein, da die $\xrightarrow{a}$-Transformation nicht im main-Thread reduzieren kann.
- Wenn $P_1 \xrightarrow{a} P_2$, dann auch $P_1 \xrightarrow{CHF} P_2$, da die $\xrightarrow{a}$-Transformation auch eine Standardreduktion ist.
- Untersuche $P_0 \xleftarrow{CHF} P_1 \xrightarrow{a} P_2$ (alle Fälle):
  Man stellt fest:

$$\begin{array}{ccc} P_1 & \xrightarrow{a} & P_2 \\ {\scriptstyle CHF}\downarrow & & \downarrow{\scriptstyle CHF} \\ P_0 & \overset{-}{\underset{a}{\rightarrow}} & P_3 \end{array}$$

Einige korrekte Programmtransformationen (2)

(1) Zeige $P_1{\downarrow_{CHF}} \implies P_2{\downarrow_{CHF}}$:

- Da $P_1{\downarrow_{CHF}}$, gibt es $P_1 \xrightarrow{CHF} P_{1,1} \xrightarrow{CHF} \ldots \xrightarrow{CHF} P_{1,n}$ mit $P_{1,n}$ erfolgreich.
- Induktion über n
- $n = 0$: P_1 ist erfolgreich. Dann muss auch P_2 erfolgreich sein. Aussage gilt.
- Induktionsschritt:



Einige korrekte Programmtransformationen (3)

(2) Zeige $P_2{\downarrow_{CHF}} \implies P_1{\downarrow_{CHF}}$:

- Gilt sofort, da die Transformation $P_1 \xrightarrow{a} P_2$ auch eine Standardreduktion ist:
- Jede konvergente Reduktionsfolge für P_2 kann durch $P_1 \xrightarrow{a} P_2$ verlängert werden zu einer konvergenten Reduktionsfolge für P_1.

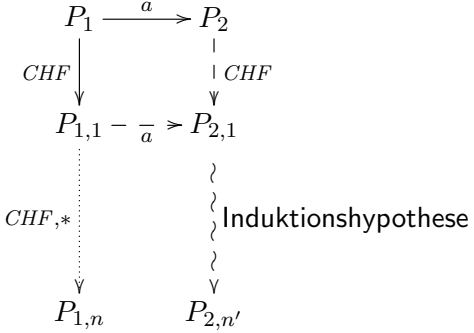
(3) Zeige $P_1{\Downarrow_{CHF}} \implies P_2{\Downarrow_{CHF}}$:

- äquivalente Aussage $P_2{\uparrow_{CHF}} \implies P_1{\uparrow_{CHF}}$
- Offensichtlich, da wiederum jede divergierende Folge für P_2 durch $P_1 \xrightarrow{a} P_2$ zu eine divergierenden Folge für P_1 verlängert werden kann.

Einige korrekte Programmtransformationen (4)

(4) Zeige $P_2 \Downarrow_{CHF} \implies P_1 \Downarrow_{CHF}$:

- äquivalente Aussage $P_1 \uparrow_{CHF} \implies P_2 \uparrow_{CHF}$:
- $P_1 \xrightarrow{CHF} P_{1,1} \xrightarrow{CHF} \ldots \xrightarrow{CHF} P_{1,n}$ wobei $P_{1,n} \Uparrow_{CHF}$
- Induktion über n
- Induktionsbasis: $P_1 \Uparrow_{CHF} \implies P_2 \Uparrow_{CHF}$: Das folgt aus (2) des Beweises!
- Indukionsschritt:



Gleichheit auf Ausdrücken

Definition

Kontextuelle Approximation $\leq_{CHF}$ und kontextuelle Gleichheit $\sim_{CHF}$ für gleich getypte Ausdrücke ist in CHF definiert also: $\leq_{CHF} := \leq_{\downarrow CHF} \cap \leq_{\Downarrow CHF}$ und $\sim_{CHF} := \leq_{CHF} \cap \geq_{CHF}$, wobei für Ausdrücke e_1, e_2 vom Typ τ:

$$e_1 \leq_{\downarrow CHF} e_2 \quad \text{gdw.} \quad \forall \mathbb{C}[\cdot^\tau] \in CC : \mathbb{C}[e_1]{\downarrow_{CHF}} \implies \mathbb{C}[e_2]{\downarrow_{CHF}}$$
$$e_1 \leq_{\Downarrow, CHF} e_2 \quad \text{gdw.} \quad \forall \mathbb{C}[\cdot^\tau] \in CC : \mathbb{C}[e_1]{\Downarrow_{CHF}} \implies \mathbb{C}[e_2]{\Downarrow_{CHF}}$$

Monadische Gesetze

Satz

In CHF gelten für alle (korrekt getypten) Ausdrücke e_1, e_2, e_3 die folgenden Gleichheiten:

$$\begin{array}{lll} \texttt{return}\ e_1\ \gg\!=\ e_2 & \sim_{CHF} & e_2\ e_1 \\ e_1\ \gg\!=\ \lambda x.\texttt{return}\ x & \sim_{CHF} & e_1 \\ e_1\ \gg\!=\ (\lambda x.(e_2\ x\ \gg\!=\ e_3)) & \sim_{CHF} & (e_1\ \gg\!=\ e_2)\ \gg\!=\ e_3 \end{array}$$

Weitere Eigenschaften von CHF

- Call-by-name Auswertung ist äquivalent zu call-by-need Auswertung
- Es wurden noch weitere Programmtransformationen als korrekt bewiesen
- CHF erweitert die pure funktionale Teilsprache konservativ:
  Alle Gleichheiten die in der puren funktionalen Sprache gelten, gelten auch in CHF
- Lazy Futures verletzen die Konservativität!