LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Prinzipien, Modelle & Algorithmen der Nebenläufigen Programmierung
Wintersemester 2020/21

Der CHF-Kalkül als Modell für Concurrent Haskell

Prof. Dr. David Sabel

LFE Theoretische Informatik

Letzte Änderung der Folien: 30. Dezember 2020

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 $x \xLeftarrow{\text{main}} e$ 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

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

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

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

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

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

- 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:

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

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

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

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

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

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

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

Typisierung: Beispiele

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

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

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

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

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

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

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

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

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

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,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) $\mathbb{L}[(\texttt{seq}\ v\ e)] \xrightarrow{CHF} \mathbb{L}[e]$

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 $\nu x_1, \ldots, x_n.x \stackrel{\text{main}}{\Longleftarrow} \texttt{return}\ e \mid P'$ 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 mit syntaktischem Zucker

```
x <=main= do future (loopPut True)   Thread, der wiederh. True in MVar x schreibt
             future (loopPut False)  Thread, der wiederh. False in MVar x schreibt
             loop
 | loop = do takeMVar x  1x Lesen
             w ← takeMVar x  1x Lesen
             if w  wenn True, dann terminiere, sonst von vorne
              then return True
              else loop
 | loopPut = λz.do putMVar x z
                   loopPut z
 | x 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. $P_1 \trianglelefteq_{\downarrow CHF} P_2$ und $P_1 \trianglelefteq_{\Downarrow CHF} P_2$, wobei

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

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

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

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

$$\begin{tikzcd} & \mathbb{D}[P_2] \arrow[dl, "CHF{,}*"'] \arrow[dr, "CHF{,}*"] & \\ \mathbb{D}'[z \xLeftarrow{\text{main}} \texttt{return True}] & & \mathbb{D}''[z \xLeftarrow{\text{main}} loop \mid loop = loop] \end{tikzcd}$$

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

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

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

$$\begin{tikzcd} & \mathbb{D}[P_2] \arrow[dl, "CHF{,}*"'] \arrow[dr, "CHF{,}*"] & \\ \mathbb{D}'[z \xLeftarrow{\text{main}} \texttt{return True}] & & \mathbb{D}''[z \xLeftarrow{\text{main}} loop \mid loop = loop] \end{tikzcd}$$

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
$(P \not\sim_{CHF} x \overset{\text{main}}{\Longleftarrow} \texttt{letrec } y = y \texttt{ in } y)$ entspricht dem Halteproblem

Ungleichheit - Beispiel

- $P_1 := x \Leftarrow \texttt{return True}$
- $P_2 := x \Leftarrow \texttt{return False}$
- $\mathbb{D} := [\cdot] \mid y \overset{\text{main}}{\Longleftarrow} \texttt{case}_{Bool}\ x$
  $(\texttt{True} \to \texttt{returnTrue})$
  $(\texttt{False} \to \texttt{letrec } w = w \texttt{ in } w)$
- $\mathbb{D}[P_1]{\downarrow}_{CHF}$ aber $\mathbb{D}[P_2]{\Uparrow}_{CHF}$.
- 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) $P_1{\downarrow}_{CHF} \implies P_2{\downarrow}_{CHF}$ (2) $P_2{\downarrow}_{CHF} \implies P_1{\downarrow}_{CHF}$

  (3) $P_1{\Downarrow}_{CHF} \implies P_2{\Downarrow}_{CHF}$ (4) $P_2{\Downarrow}_{CHF} \implies P_1{\Downarrow}_{CHF}$

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:

$$\begin{tikzcd} P_1 \arrow[r, "a"] \arrow[d, "CHF"'] & P_2 \arrow[d, dashed, "CHF"] \\ P_0 \arrow[r, dashed, "a"'] & P_3 \end{tikzcd}$$

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:

$$\begin{tikzcd} P_1 \arrow[r, "a"] \arrow[d, "CHF"'] & P_2 \arrow[d, dashed, "CHF"] \\ P_{1,1} \arrow[r, dashed, "a"'] \arrow[d, dotted, "{CHF,*}"'] & P_{2,1} \arrow[d, squiggly, "\text{Induktionshypothese}"] \\ P_{1,n} & P_{2,n'} \end{tikzcd}$$

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:

$$\begin{tikzcd}
P_1 \arrow[r, "a"] \arrow[d, "CHF"'] & P_2 \arrow[d, dashed, "CHF"] \\
P_{1,1} \arrow[r, dashed, "a"'] \arrow[d, dotted, "{CHF,*}"'] & P_{2,1} \arrow[d, squiggly, "\text{Induktionshypothese}"] \\
P_{1,n} & P_{2,n'}
\end{tikzcd}$$

Gleichheit auf Ausdrücken

Definition

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

$$\begin{array}{lll}
e_1 \trianglelefteq_{\downarrow CHF} e_2 & \text{gdw.} & \forall \mathbb{C}[\cdot^\tau] \in CC : \mathbb{C}[e_1]{\downarrow}_{CHF} \implies \mathbb{C}[e_2]{\downarrow}_{CHF} \\
e_1 \trianglelefteq_{\Downarrow, CHF} e_2 & \text{gdw.} & \forall \mathbb{C}[\cdot^\tau] \in CC : \mathbb{C}[e_1]{\Downarrow}_{CHF} \implies \mathbb{C}[e_2]{\Downarrow}_{CHF}
\end{array}$$

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!