

## Der CHF-Kalkül als Modell für Concurrent Haskell

Prof. Dr. David Sabel

LFE Theoretische Informatik



Letzte Änderung der Folien: 6. Februar 2020

### 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

### Übersicht

- 1 Der CHF-Kalkül
  - Syntax
  - Typisierung
  - Semantik
  - Fairness
  - Gleichheit

### Der CHF-Kalkül: Syntax

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

$P, P_i \in Proc$	$::=$	$P_1 \mid P_2$	parallele Komposition
		$\nu x.P$	Namensbeschränkung
		$x \leftarrow e$	nebenl. Thread (Future $x$ )
		$x = e$	globale Bindung
		$x \mathbf{m} e$	gefüllte MVar
		$x \mathbf{m} -$	leere MVar

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

Es gibt Datenkonstruktoren  $c_{T,i}$

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

$$\text{case}_T e \text{ of } \underbrace{(c_{T,1} x_1 \dots x_{ar(c_{T,1})} \rightarrow e_1)}_{\text{case-Alternative}} \dots \underbrace{(c_{T,|T|} x_1 \dots x_{ar(c_{T,|T|})} \rightarrow e_{|T|})}_{\text{Pattern}}$$

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

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

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

## Funktionale Ausdrücke

$e, e_i \in \text{Exp} ::= x$	Variable
$me$	monad. Ausdruck
$\lambda x.e$	Abstraktion
$(e_1 e_2)$	Anwendung
$c e_1 \dots e_{ar(c)}$	Konstruktoranw.
$\text{seq } e_1 e_2$	seq-Ausdruck
$\text{letrec } x_1 = e_1, \dots, x_n = e_n \text{ in } e$	letrec-Ausdruck
$\text{case}_T e \text{ of } (c_{T,1} x_1 \dots x_{ar(c_{T,1})} \rightarrow e_1) \dots (c_{T, T } x_1 \dots x_{ar(c_{T, T })} \rightarrow e_{ T })$	case-Ausdruck

## Monadische Ausdrücke

$$m \in \text{MExp} ::= \text{return } e \mid e_1 \gg e_2 \mid \text{future } e \mid \text{takeMVar } e \mid \text{newMVar } e \mid \text{putMVar } e_1 e_2$$

- **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 kleinste 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
- $\tau, \tau_i \in Typ ::= IO \tau \mid (T \tau_1 \dots \tau_{ar(T)}) \mid MVar \tau \mid \tau_1 \rightarrow \tau_2$
- 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  $\Gamma(x) \in Typ$
- $\Gamma \vdash P :: wt$  gdw. Prozess  $P$  ist wohlgetypt
- $\Gamma \vdash e :: \tau$  gdw. Ausdruck  $e$  ist wohlgetypt mit Typ  $\tau$ .

## Typisierung: Beispiele

$$\frac{\Gamma(x) = \tau \quad \Gamma(x) = \tau}{\Gamma \vdash x :: \tau}, \frac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau}$$

$$\Gamma \vdash (\lambda x.x) :: \tau \rightarrow \tau$$

$$\frac{\Gamma(x) = \tau \quad \Gamma(x) = \tau}{\Gamma \vdash x :: \tau}, \frac{\Gamma(x) = \tau \quad \Gamma(x) = \tau}{\Gamma \vdash x :: \tau}, \frac{\Gamma \vdash e_1 :: \tau_1 \rightarrow \tau_2, \Gamma \vdash e_2 :: \tau_1}{\Gamma \vdash (e_1 e_2) :: \tau_2}$$

$$\frac{\Gamma \vdash x :: \tau, \Gamma \vdash x :: \tau}{\Gamma \vdash (x x) :: ?} \quad \frac{\Gamma \vdash e_1 :: \tau_1 \rightarrow \tau_2, \Gamma \vdash e_2 :: \tau_1}{\Gamma \vdash (e_1 e_2) :: \tau_2}$$

$$\lambda x.(x x) :: \tau \rightarrow ?$$

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

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

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

## Typisierung (2)

Einige Typisierungsregeln

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

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

$$\frac{\Gamma \vdash e :: \tau}{\Gamma \vdash \text{return } e :: IO \tau} \quad \frac{\Gamma \vdash e_1 :: IO \tau_1, \Gamma \vdash e_2 :: \tau_1 \rightarrow IO \tau_2}{\Gamma \vdash e_1 \gg e_2 :: IO \tau_2}$$

$$\frac{\Gamma \vdash e :: IO \tau}{\Gamma \vdash \text{future } e :: IO \tau} \quad \frac{\Gamma \vdash e :: MVar \tau}{\Gamma \vdash \text{takeMVar } e :: IO \tau}$$

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

## Operationale Semantik: Kontexte

**Prozesskontexte:**

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

**Monadische Kontexte:**

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

**Evaluations- und Forcingkontexte**

$$\mathbb{E} \in EC ::= [\cdot] \mid (\mathbb{E} e) \mid (\text{case } \mathbb{E} \text{ of } \text{alts}) \mid (\text{seq } \mathbb{E} e)$$

$$\mathbb{F} \in FC ::= \mathbb{E} \mid (\text{takeMVar } \mathbb{E}) \mid (\text{putMVar } \mathbb{E} e)$$

$\mathbb{L} \in LC ::= x \Leftarrow M[\mathbb{F}]$   
 $| x \Leftarrow M[\mathbb{F}[x_n] \mid x_n = \mathbb{E}_n[x_{n-1}] \mid \dots \mid x_2 = \mathbb{E}_2[x_1] \mid x_1 = \mathbb{E}_1]$   
 wobei  $\mathbb{E}_2, \dots, \mathbb{E}_n$  nicht der leere Kontext sind.

$\widehat{\mathbb{L}} \in \widehat{LC} ::= x \Leftarrow M[\mathbb{F}]$   
 $| x \Leftarrow M[\mathbb{F}[x_n] \mid x_n = \mathbb{E}_n[x_{n-1}] \mid \dots \mid x_2 = \mathbb{E}_2[x_1] \mid x_1 = \mathbb{E}_1]$   
 wobei  $\mathbb{E}_1, \mathbb{E}_2, \dots, \mathbb{E}_n$  nicht der leere Kontext sind.

## Standardreduktion $\xrightarrow{CHF}$ (2)

### Funktionale Auswertung:

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

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

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

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

$(CHF, case) \quad \mathbb{L}[\text{case}_T (c e_1 \dots e_n) \text{ of } \dots (c y_1 \dots y_n \rightarrow e) \dots]$   
 $\xrightarrow{CHF} \nu y_1, \dots, y_n. (\mathbb{L}[e \mid y_1 = e_1 \mid \dots \mid y_n = e_n])$

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

### Monadische Berechnungen:

$(CHF, \text{lunit}) \quad y \Leftarrow M[\text{return } e_1 \gg e_2] \xrightarrow{CHF} y \Leftarrow M[e_2 e_1]$

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

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

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

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

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

## Standardreduktion: Abschluss

$$\frac{P_1 \equiv \mathbb{D}[P'_1], P_2 \equiv \mathbb{D}[P'_2] \text{ und } P'_1 \xrightarrow{CHF} P'_2}{P_1 \xrightarrow{CHF} P_2}$$

### Definition

Ein wohlgeformter Prozess  $P$  ist genau dann **erfolgreich**, wenn er von der Form  $\nu x_1, \dots, x_n. x \xrightarrow{\text{main}} \text{return } e \mid P'$  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)

## Should-Konvergenz $\neq$ Must-Konvergenz

Beispiel mit syntaktischem Zucker

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

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

|  $\text{loopPut} = \lambda z. \text{do putMVar } x \ z$   
 $\text{loopPut } z$

|  $x \text{ m -}$

Prozess ist should-konvergent, aber nicht must-konvergent

### 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

## 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.  $P_1 \leq_{CHF} P_2$  und  $P_1 \leq_{\downarrow CHF} P_2$ , wobei

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

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

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

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

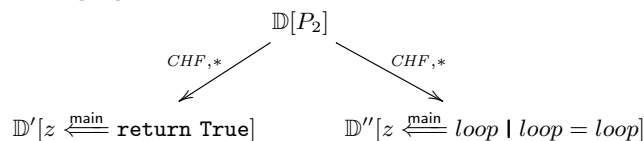
## Should-Konvergenz alleine reicht nicht

$P_1 := \nu z, loop. (z \stackrel{\text{main}}{\longleftarrow} loop) \mid loop = loop$

$P_2 := \nu x, z, y_1, y_2, loop.$

$(z \stackrel{\text{main}}{\longleftarrow} \text{takeMVar } x \gg \lambda w. \text{case}_{Bool} w (\text{True} \rightarrow \text{return True}) (\text{False} \rightarrow loop)$   
 $\mid loop = loop \mid y_1 \leftarrow \text{putMVar } x \text{ False} \mid y_2 \leftarrow \text{putMVar } x \text{ True} \mid x \mathbf{m} -)$

- $\mathbb{D}[P_1]$  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  $P_2 \downarrow_{CHF}$

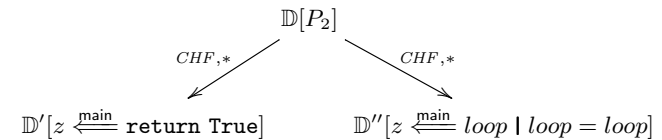
## May-Konvergenz alleine reicht nicht

$P_1 := \nu z. (z \stackrel{\text{main}}{\longleftarrow} \text{return True})$

$P_2 := \nu x, z, y_1, y_2, loop.$

$(z \stackrel{\text{main}}{\longleftarrow} \text{takeMVar } x \gg \lambda w. \text{case}_{Bool} w (\text{True} \rightarrow \text{return True}) (\text{False} \rightarrow loop)$   
 $\mid loop = loop \mid y_1 \leftarrow \text{putMVar } x \text{ False} \mid y_2 \leftarrow \text{putMVar } x \text{ True} \mid x \mathbf{m} -)$

- $\mathbb{D}[P_1]$  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  $P_2 \uparrow_{CHF}$

## Fairness

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

Beispiel:

$$x \stackrel{\text{main}}{\longleftarrow} \text{takeMVar } z \mid z \mathbf{m} \text{ True} \mid y \leftarrow loop \mid loop = loop$$

$$\xrightarrow{CHF, cp} x \stackrel{\text{main}}{\longleftarrow} \text{takeMVar } z \mid z \mathbf{m} \text{ True} \mid y \leftarrow loop \mid loop = loop$$

$$\xrightarrow{CHF, cp} x \stackrel{\text{main}}{\longleftarrow} \text{takeMVar } z \mid z \mathbf{m} \text{ True} \mid y \leftarrow loop \mid loop = loop$$

$$\xrightarrow{CHF, cp} x \stackrel{\text{main}}{\longleftarrow} \text{takeMVar } z \mid z \mathbf{m} \text{ True} \mid y \leftarrow loop \mid loop = loop$$

$$\xrightarrow{CHF, cp} \dots$$

## 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  $S = P \xrightarrow{CHF} P_1 \xrightarrow{CHF} P_2 \dots$  **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.

## 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

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

## 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!

## Ungleichheit - Beispiel

- $P_1 := x \leftarrow \text{return True}$
- $P_2 := x \leftarrow \text{return False}$
- $\mathbb{D} := [\cdot] \mid y \xleftarrow{\text{main}} \text{case}_{\text{Bool}} x$   
(True  $\rightarrow$  return True)  
(False  $\rightarrow$  letrec  $w = w$  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

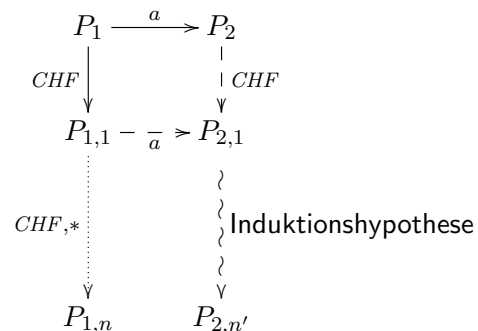
Die Reduktionen  $(CHF, \text{unit})$ ,  $(CHF, \text{nmvar})$ ,  $(CHF, \text{fork})$ ,  $(CHF, \text{unIO})$ ,  $(CHF, \text{mkbinds})$  sind korrekte Programmtransformationen.

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 (2)

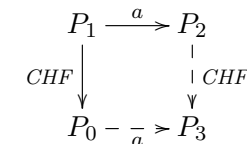
- (1) Zeige  $P_1 \downarrow_{CHF} \implies P_2 \downarrow_{CHF}$ :
  - Da  $P_1 \downarrow_{CHF}$ , gibt es  $P_1 \xrightarrow{CHF} P_{1,1} \xrightarrow{CHF} \dots \xrightarrow{CHF} P_{1,n}$  mit  $P_{1,n}$  erfolgreich.
  - Induktion über  $n$
  - $n = 0$ :  $P_1$  ist erfolgreich. Dann muss auch 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



## Einige korrekte Programmtransformationen

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:



## Einige korrekte Programmtransformationen (3)

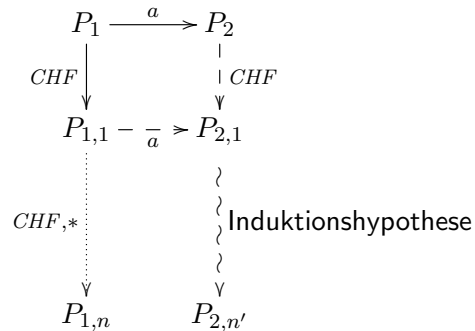
- (2) Zeige  $P_2 \downarrow_{CHF} \implies P_1 \downarrow_{CHF}$ :
  - Gilt sofort, da die Transformation  $P_1 \xrightarrow{a} P_2$  auch eine Standardreduktion ist:
  - Jede konvergente Reduktionsfolge für  $P_2$  kann durch  $P_1 \xrightarrow{a} P_2$  verlängert werden zu einer konvergenten Reduktionsfolge 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  $P_1 \Uparrow_{CHF} \implies P_2 \Uparrow_{CHF}$ :
- $P_1 \xrightarrow{CHF} P_{1,1} \xrightarrow{CHF} \dots \xrightarrow{CHF} P_{1,n}$  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!
- Induktionsschritt:



## Gleichheit auf Ausdrücken

### Definition

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

$$\begin{array}{ll}
 e_1 \leq_{CHF} e_2 & \text{gdw. } \forall C[\cdot, \tau] \in CC : C[e_1] \Downarrow_{CHF} \implies C[e_2] \Downarrow_{CHF} \\
 e_1 \sim_{CHF} e_2 & \text{gdw. } \forall C[\cdot, \tau] \in CC : C[e_1] \Downarrow_{CHF} \implies C[e_2] \Downarrow_{CHF}
 \end{array}$$

## Monadische Gesetze

### Satz

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

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

## 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!