GOETHE UNIVERSITÄT FRANKFURT AM MAIN

www.uni-frankfurt.de

Korrektheit von Programmen und Programmiersprachen

Methoden, Analysen und Anwendungen

David Sabel

Goethe-Universität, Frankfurt am Main

13. November 2012

Überblick

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

1. **Motivation**
2. **Grundlagen**
3. **Korrektheit von Übersetzungen und Implementierungen**
4. **Semantik von Concurrent Haskell mit Futures**
5. **Eine Logik für eine funktionale call-by-value Sprache**
6. **Fazit**

Motivation

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Ziel: Korrekte Programme / Systeme

- Schäden durch fehlerhafte Software vermeiden
- Einzug von informatischen Systemen in immer mehr Alltagsgegenstände
- Verantwortung der Informatik steigt!

Motivation (2)

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Bereiche:

- Korrekte Compiler (Semantikerhaltend!)
- Korrekte Bibliotheksfunktionen
- Techniken / Methoden zur Verifikation von Programmen
- Korrektheitsnachweise für realistische, moderne (insbes. funktionale, nebenläufige) Programmiersprachen

Anwendungsbereiche

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Korrektheit von Programmtransformationen

```
for (X:=1, X < n, X++) {
  Z := Z + X;
}
```

→

```
X:=1
while X < n {
  Z := Z + X;
  X++;
}
```

Programmtransformation $\xrightarrow{T}$: binäre Relation auf Programmfragmenten,

$\xrightarrow{T}$ ist **korrekt** gdw. $P_1 \xrightarrow{T} P_2 \implies P_1$ semantisch äquivalent zu P_2

Beispiele:

- Compiler: Optimierungen (Garbage Collection, Procedure Inlining, partielles Auswerten. . .)
- „Program Refactoring“: Transformationen zur Verbesserung der Les- und Wartbarkeit
- Theorem-Beweiser: Umformungen von Programmen in Beweisen

Anwendungsbereiche (2)

Korrekte Übersetzungen zwischen Programmiersprachen

```
map f []     = []
map f (x:xs) = (f x):(map f xs)
```

→

```
01010101110110101001
11110101010101010101
```

Beispiele:

- Compiler: Übersetzung von Quell- in Zielsprache
- Vergleich der Ausdruckskraft von Sprachen / Kalkülen
- Korrektheit von Implementierungen (z.B. Semaphore durch Locks)

Anwendungsbereiche (3)

Eigenschaften von Programmen

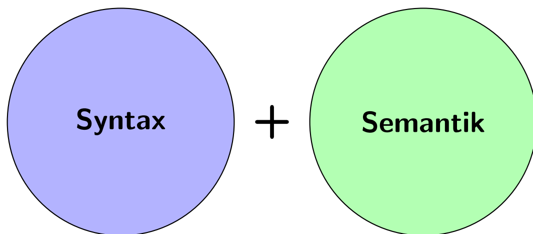
```
reverse []     = []
reverse (x:xs) = (reverse xs) ++ [x]
```

Gilt $\forall xs :: \text{Liste}.(\texttt{reverse}\ (\texttt{reverse}\ xs)) = xs$?

- Gültigkeit von Aussagen über Programmen
- Formulierung der Eigenschaften in einer Logik (Spezifikationssprache)
- Tautologische Formeln $=$ Korrekte Aussagen
- (Halb-)automatische Theorembeweiser

Grundlagen

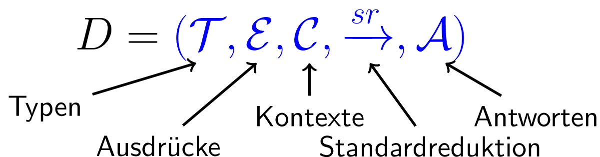
Programmiersprachen



- **Syntax**: Festlegung der gültigen Programmen
  (üblicherweise durch CFG + Nebenbedingungen)
- **Semantik**: Bedeutung der Programme
  Operationale Semantik, d.h. wie ein Programm ausgeführt wird
- Verschiedene Formalismen zur operationalen Semantik: Zustandsübergangssysteme, Ersetzungssysteme, Abstrakte Maschinen, ...

Rahmen für Programmiersprachen

Getypter Programmkalkül (mit small-step Reduktion)

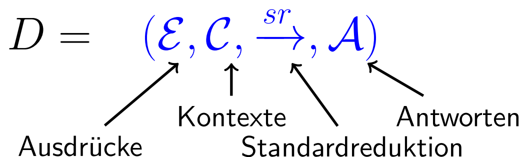


- Ausdrücke $e \in \mathcal{E}$ sind getypt mit Typen $T \in \mathcal{T}$
- Kontexte $C \in \mathcal{C}$ sind Funktionen $C : \mathcal{E} \to \mathcal{E}$
  üblicherweise: „alle Ausdrücke mit Loch $[\cdot]$“
- Standardreduktion $\xrightarrow{sr} \subseteq (\mathcal{E} \times \mathcal{E})$ = operationale Semantik
- Antworten $\mathcal{A} \subseteq \mathcal{E}$ i.A. irreduzibel (Werte, Normalformen)
- Deterministisch gdw. $e_1 \xrightarrow{sr} e_2 \wedge e_1 \xrightarrow{sr} e_3 \implies e_2 = e_3$

Rahmen für Programmiersprachen

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Ungetypter Programmkalkül (mit small-step Reduktion)



- ~~Ausdrücke $\mathcal{E}$ sind getypt mit Typen aus $\mathcal{T}$~~
- Kontexte $C \in \mathcal{C}$ sind Funktionen $C : \mathcal{E} \to \mathcal{E}$
  üblicherweise: „alle Ausdrücke mit Loch $[\cdot]$“
- Standardreduktion $\xrightarrow{sr} \subseteq (\mathcal{E} \times \mathcal{E})$ = operationale Semantik
- Antworten $\mathcal{A} \subseteq \mathcal{E}$ i.A. irreduzibel (Werte, Normalformen)
- Deterministisch gdw. $e_1 \xrightarrow{sr} e_2 \wedge e_1 \xrightarrow{sr} e_3 \quad \Longrightarrow \quad e_2 = e_3$

Beispiele

Lazy Lambda-Kalkül: $D = (\mathcal{E}_\lambda, \mathcal{C}_\lambda, \xrightarrow{lazy}, \mathcal{A}_\lambda)$

- Ausdrücke $e \in \mathcal{E}_\lambda ::= x \mid \lambda x.e \mid (e_1\ e_2)$
- Kontexte $C \in \mathcal{C}_\lambda ::= [\cdot] \mid (C\ e) \mid (e\ C) \mid \lambda x.C$
- Standardreduktion (wobei $R \in \mathcal{R} ::= [\cdot] \mid (R\ e)$):
  $R[(\lambda x.e_1)\ e_2] \xrightarrow{lazy} R[e_1[e_2/x]]$
- Antworten $\mathcal{A}_\lambda := \{\lambda x.e \mid e \in \mathcal{E}_\lambda\}$

Beispielreduktion:

$(\lambda x.x\ x)\ ((\lambda u.u)\ \lambda z.z)$
$\xrightarrow{lazy} ((\lambda u.u)\ (\lambda z.z))\ ((\lambda u'.u')\ \lambda z'.z')$
$\xrightarrow{lazy} (\lambda z.z)\ ((\lambda u'.u')\ \lambda z'.z') \xrightarrow{lazy} ((\lambda u'.u')\ \lambda z'.z') \xrightarrow{lazy} \lambda z'.z'$

Beispiele (2)

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Lazy Lambda-Kalkül mit Choice: $D = (\mathcal{E}_\oplus, \mathcal{C}_\oplus, \xrightarrow{lazynd}, \mathcal{A}_\oplus)$

- Ausdrücke $e \in \mathcal{E}_\oplus ::= \ldots \mid e_1 \oplus e_2$
- Kontexte $C \in \mathcal{C}_\oplus ::= \ldots \mid C \oplus e \mid e \oplus C$
- Standardreduktion (wobei $R \in \mathcal{R} ::= [\cdot] \mid (R\ e)$):
  - $R[(\lambda x.e_1)\ e_2] \xrightarrow{lazynd} R[e_1[e_2/x]]$
  - $R[e_1 \oplus e_2] \xrightarrow{lazynd} R[e_i]$ mit $i = 1 \lor i = 2$
- Antworten $\mathcal{A}_\oplus := \{\lambda x.e \mid e \in \mathcal{E}\}$

Gleichheitssemantik von Programmen (1)

Wann soll $e_1 = e_2$ gelten?

Anforderungen an einen Gleichheitsbegriff

- kanonisch: auf viele Programmiersprachen anwendbar
- grobkörnig: möglichst viele Gleichheiten, keine unnötigen Einschränkungen
- Äquivalenzrelation: offensichtlich
- Kompatibel mit Kontexten:
  $e_1 = e_2 \implies C[e_1] = C[e_2]$
  - lokale Anwendbarkeit der Gleichheit
  - z.B. Separate Kompilierung von Modulen!

(Äquivalenzrelation + Kompatibel mit Kontexten) } Kongruenz

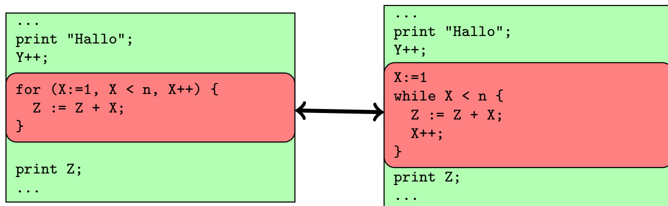
Gleichheitssemantik von Programmen (2)

Einige Ansätze:

- Syntaktische Gleichheit: viel zu fein
- Syntaktisch gleiche Antworten: i.A. viel zu fein, falsch bei Nichtdeterminismus
- Syntaktisch ineinander überführbar = Konvertibilität: zu fein, falsch bei Nichtdeterminismus
- Denotationale Semantik = gleiche Denotation: eher nicht kanonisch
- Gleiches Verhalten in allen Kontexten

Kontextuelle Gleichheit

Ausdrücke / Programme e_1 und e_2 sind **kontextuell gleich**, wenn der Austausch von e_1 durch e_2 an beliebiger Position nicht beobachtbar ist



e_1 und e_2 verhalten sich in allen Kontexten gleich.

Verhalten:
Bei determinstischen Kalkülen: Beobachtung der Terminierung

Kontextuelle Gleichheit

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

May-Konvergenz

$e \in \mathcal{E}$ may-konvergiert $(e{\downarrow_D})$ gdw. $\exists v \in \mathcal{A} : e \xrightarrow{sr,*} v$

$(\neg e{\downarrow_D} =$ „e must-divergiert“, $e{\Uparrow_D})$

Kontextuelle Präordnung und Gleichheit

(für deterministische Kalküle):

- $e_1 \leq_{\downarrow} e_2$ gdw. $\forall C : C[e_1]{\downarrow} \implies C[e_2]{\downarrow}$
- $e_1 \sim_{\downarrow} e_2$ gdw. $e_1 \leq_{\downarrow} e_2 \quad \wedge \quad e_2 \leq_{\downarrow} e_1$

Eigenschaften

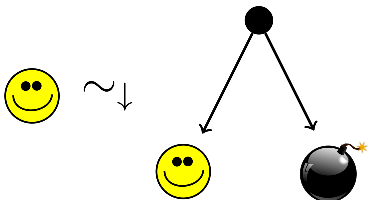
GOETHE UNIVERSITÄT FRANKFURT AM MAIN

- $\sim_\downarrow$ ist **kanonisch**: für jede Beschreibung eines Programmkalküls
- $\sim_\downarrow$ ist i.A. eine **Kongruenz** (bei Typisierung muss man aufpassen)
- **sehr grobkörnige** Gleichheit
- $\sim_\downarrow$ und $\not\sim_\downarrow$ sind i.A. **unentscheidbar** ($e \not\sim_\downarrow \Omega$ = Halteproblem)
- Gleichheit widerlegen ($e_1 \not\sim_\downarrow e_2$): **Ein Kontext** als Gegenbeispiel
  Z.B. $1 \not\sim_\downarrow 2$: für $C := \texttt{if } [\cdot] = 1 \texttt{ then } 1 \texttt{ else } \Omega$ gilt $C[1]\downarrow$ und $C[2]\Uparrow$
- Gleichheit zeigen ($e_1 \sim_\downarrow e_2$): **Alle Kontexte** betrachten
  Hilfsmittel z.B. Kontextlemmata (z.B. [SSS10,TCS]), Diagrammmethode (z.B. [RSSS12,IJCAR]), Bisimulation (z.B. [SSSM10,RTA]; [SSSM11,IPL])

Kontextuelle Gleichheit bei Nichtdeterminismus

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Betrachtung der May-Konvergenz reicht nicht aus:



- Programme mit möglicher Fehlersituation (Divergenz, Deadlock, ...) werden mit fehlerlosen Programmen gleichgesetzt
- Z.B. $\lambda x.x \sim_\downarrow \lambda x.x \ \oplus \ ((\lambda x.x\ x)\ (\lambda x.x\ x))$

Kontextuelle Gleichheit bei Nichtdeterminismus (2)

Ausweg: Zusätzlich Should-Konvergenz betrachten

Should-Konvergenz

$e \in \mathcal{E}$ should-konvergiert ($e{\Downarrow_D}$) gdw. $\forall e' \in \mathcal{E} : e \xrightarrow{sr,*} e' \implies e{\downarrow_D}$

($\neg e{\Downarrow_D}$ = „e may-divergiert“, $e{\uparrow_D}$, äquiv. zu $\exists e' : e \xrightarrow{sr,*} e' \wedge e'{\Uparrow_D}$)

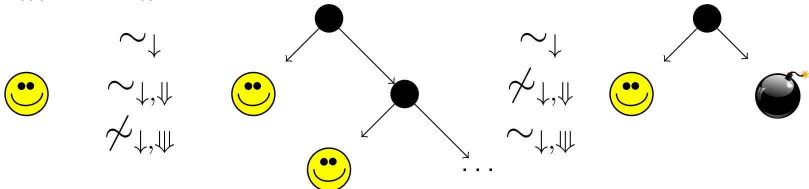
Kontextuelle Präordnung und Gleichheit

- $e_1 \leq_{\Downarrow} e_2$ gdw. $\forall C : (C[e_1]{\Downarrow_D} \implies C[e_2]{\Downarrow_D})$.
- $e_1 \leq_{\downarrow,\Downarrow} e_2$ gdw. $e_1 \leq_{\downarrow} e_2 \wedge e_1 \leq_{\Downarrow} e_2$
- $e_1 \sim_{\downarrow,\Downarrow} e_2$ gdw. $e_1 \leq_{\downarrow,\Downarrow} e_2 \wedge e_2 \leq_{\downarrow,\Downarrow} e_1$

Should- vs. Must-Konvergenz

Must-Konvergenz: $e{\Downarrow\!\!\Downarrow}_D$ gdw. $e{\Downarrow} \land \neg(e \xrightarrow{sr,\omega})$

$\sim_{\downarrow,\Downarrow}$ und $\sim_{\downarrow,\Downarrow\!\!\Downarrow}$ sind leicht verschieden:



Z.B. $e_1 := \mathbf{I}$ $\quad e_2 := \mathbf{Y}\ \lambda f.(\mathbf{I} \oplus f)$ $\quad e_3 := (\mathbf{I} \oplus \mathbf{\Omega})$

Vorteile der May/Should-Kombination

- Explizite Fairnessbetrachtung nicht erforderlich
- Should erkennt busy-waiting nicht als Fehler
- Induktionsbeweise für Should-Konvergenz möglich
- Abgeschlossen bezüglich weiterer Kombinationen [SSS10,IPL]

Ziel

Untersuchung der verschiedenen Anwendungsbereiche zur Korrektheit von Programmen, Implementierungen, Übersetzungen und Eigenschaften von Programmen mit

kontextueller Äquivalenz als Gleichheitssemantik

Forschungsarbeiten: Themenblöcke

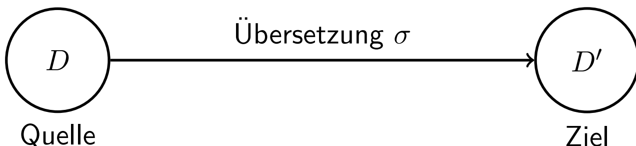
- **Korrektheit von Übersetzungen und Implementierungen**
  [NSSSS08,IFIP TCS] & [SSSSN09,ML]
- **Semantik von nebenläufigen Programmiersprachen am Beispiel Concurrent Haskell mit Futures**
  [SSS11,PPDP], [DS12,ATPS] & [SSS12,LICS]
- **Eine zweiwertige Logik für eine funktionale call-by-value Sprache mit kontextueller Äquivalenz**
  [SSS12,JAR]

- **Semantik von call-by-need Letrec-Kalkülen: Bisimulation, Isomorphie, Typisierung**
  [SSSH09,ATPS], [SSSM10,RTA] & [SSSM11,IPL]
- **Hilfsmittel zum Nachweis kontextueller Äquivalenz**
  [SSS10,TCS] & [RSSS12,IJCAR]

Korrektheit von Übersetzungen und Implementierungen

Übersetzungen [SSNSS08,IFIP TCS]

Übersetzung zwischen Programmkalkülen, $\sigma : D \to D'$



- bildet D-Typen auf D'-Typen ab
- bildet D-Ausdrücke auf D'-Ausdrücke ab
- bildet D-Kontexte auf D'-Kontexte ab
- D und D' haben die gleichen Beobachtungsprädikate

Randbedingungen

- σ ist typgerecht
- $\sigma([\cdot]) = [\cdot]$

Eigenschaften von Übersetzungen

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

σ ist ...

- **adäquat** gdw. $\forall e_1, e_2 : \sigma(e_1) \leq_{D'} \sigma(e_2) \implies e_1 \leq_D e_2$
- **voll abstrakt** gdw. $\forall e_1, e_2 : \sigma(e_1) \leq_{D'} \sigma(e_2) \iff e_1 \leq_D e_2$
- **kompositional bzgl. Konvergenz (CUO)** gdw.
  $\forall C, e, i : \sigma(C[e])\Downarrow'_i \iff \sigma(C)\sigma(e)\Downarrow'_i$
- **konvergenzäquivalent (CE)** gdw. $\forall e : e\Downarrow_i \iff \sigma(e)\Downarrow'_i$.
- **beobachtungskorrekt** gdw. CUO & CE, d.h.
  $\forall e, C, i : C[e]\Downarrow_i \iff \sigma(C)[\sigma(e)]\Downarrow'_i$.

Satz [SSNSS08,IFIP TCS]

Jede beobachtungskorrekte Übersetzung ist auch adäquat.

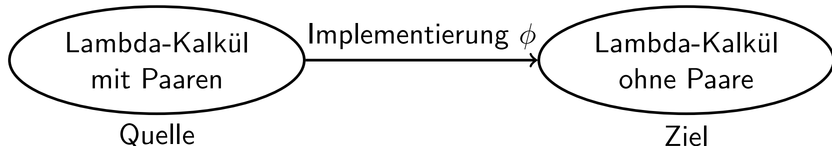
Korrektheit je nach Anwendung

- Kompilieren: Adäquatheit (Beobachtungskorrektheit): Transformationen nach der Übersetzung sind auch korrekt aus Sicht der Quellsprache
- Reines Interpretieren: Konvergenzäquivalenz
- Stufenweises Übersetzen: Unproblematisch, alle Eigenschaften setzen sich fort über Komposition
- Implementierung von Sprachkonstrukten: Übersetzung ist beobachtungskorrekt (adäquat), d.h. keine neuen Gleichheiten durch die Implementierung

Didaktisches Beispiel: Paar-Kodierung von Church

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

$$\begin{array}{lcl} \phi(v_1, v_2) & := & \lambda x.(x\ \phi(v_1)\ \phi(v_2)) \\ \phi(\mathbf{fst}) & := & \lambda p.(p\ (\lambda x, y.x)) \\ \phi(\mathbf{snd}) & := & \lambda p.(p\ (\lambda x, y.y)) \end{array}$$



Ergebnisse aus [SSNSS08, IFIP TCS]:

- ϕ nur dann adäquat/beobachtungskorrekt wenn Quelle getypt & Ziel ungetypt
- Weitere Resultate bzgl. nicht-deterministischer Erweiterungen

Realistisches Beispiel [SSSSN09, ML]

Synchronisierende **Puffer**:
leer oder voll
$x \, \mathsf{b} \, -$ und $x \, \mathsf{b} \, v$
get(x) und **put**(x, w)

Handles:
$x \, \mathsf{h} \, y$ und $x \, \mathsf{h} \, \bullet$
einmalig binden
$x \, \mathsf{h} \, y \ldots \rightarrow x \, \mathsf{h} \, \bullet \;\; | \;\; y \Leftarrow \; v$

call-by-value
Prozesskalkül
mit Futures
und Speicherzellen

Alice ML

Sind Puffer **korrekt** implementierbar?

Realistisches Beispiel [SSSSN09, ML]

Synchronisierende **Puffer**:
leer oder voll
$x \,\mathsf{b}\, -$ und $x \,\mathsf{b}\, v$
get(x) und **put**(x, w)

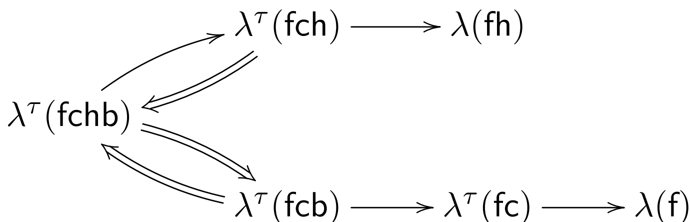
Handles:
$x \,\mathsf{h}\, y$ und $x \,\mathsf{h}\, \bullet$
einmalig binden
$x \,\mathsf{h}\, y \ldots \rightarrow x \,\mathsf{h}\, \bullet \;\mid\; y \Leftarrow v$

call-by-value
Prozesskalkül
mit Futures
und Speicherzellen

Sind Handles **korrekt** implementierbar?

Realistisches Beispiel (2)

Resultat aus [SSSSN09, ML]



$\longrightarrow$ Adäquate Übersetzung

$\Longrightarrow$ Voll-abstrakte Übersetzung

- bzgl. kontextueller Äquivalenz mit May- und Should-Konvergenz
- Außerdem: Case & Konstruktoren weg kodierbar
- Außerdem: Handles und Puffer implementierbar (mit busy-wait)

Semantik von nebenläufigen Programmiersprachen am Beispiel Concurrent Haskell mit Futures

Motivation: Haskell

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Funktionale Kernsprache
call-by-need Lambda-Kalkül
mit letrec, seq, Datenkonstruktoren

+ **Monadisches I/O** ≈ **Haskell**

+ **nebenläufige Threads & MVars**
≈ **Concurrent Haskell**

+ **lazy I/O** unsafePerformIO, unsafeInterleaveIO
≈ **Echte Implementierungen von Haskell**

- semantisch **gut verstanden** & ausführlich **untersucht**
- viele **korrekte** Programmtransformationen und Optimierungen

Der CHF-Kalkül

- Kernsprache für Concurrent Haskell erweitert um Futures
- Nebenläufige Threads
- Synchronisierende Puffer (MVars)
- Kapselung von Seiteneffekten durch monadisches I/O

Syntax

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Prozesse

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

Es gibt einen Main-Thread: $x \overset{\text{main}}{\Longleftarrow} e \,|\, P$

Ausdrücke

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

$$
\begin{aligned}
me \in \mathit{MExpr} ::= {} & \texttt{return}\ e \mid e_1 \gg\!= e_2 \mid \texttt{future}\ e \\
& \mid \texttt{takeMVar}\ e \mid \texttt{newMVar}\ e \mid \texttt{putMVar}\ e_1\ e_2
\end{aligned}
$$

Operationale Semantik

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

- Small-step Reduktion $\xrightarrow{CHF}$ auf Prozessen
- 13 Reduktionsregeln

Auszug:

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

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

$\mathbb{L}$-Kontexte: $\mathbb{L} ::= x \Leftarrow \mathbb{M}[\mathbb{F}]$
$\mid\ x \Leftarrow \mathbb{M}[\mathbb{F}[x_n]] \mid x_n = \mathbb{E}_n[x_{n-1}] \mid \ldots \mid x_2 = \mathbb{E}_2[x_1] \mid x_1 = \mathbb{E}_1$

Evaluationskontexte: $\mathbb{E} ::= [\cdot] \mid (\mathbb{E}\ e) \mid (\texttt{case}\ \mathbb{E}\ \texttt{of}\ alts) \mid (\texttt{seq}\ \mathbb{E}\ e)$

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

Korrekte Programmtransformationen [SSS11,PPDP]

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

In [SSS11, PPDP] wurde gezeigt (bzgl. $\sim_{\downarrow,\Downarrow}$):

Theorem

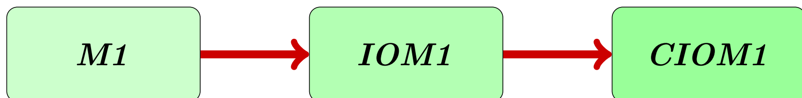
- Alle Regeln zur funktionalen Auswertung sind korrekt als Transformationen
- Alle Standardreduktionen sind korrekt (bis auf take und put)
- Die monadischen Gesetze gelten
- Weitere Optimierungen sind korrekte Programmtransformationen:
  Kopieren, Garbage Collection, ...

Außerdem: Call-by-name Auswertung ist äquivalent bzgl. $\downarrow$ und $\Downarrow$

Abstrakte Maschine [DS12,ATPS]

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Dreistufige Herleitung einer abstrakten Maschine für CHF:



Zustand (*CIOM1*): (Heap,MVars,Threads)
wobei Thread = (Name,Ausdruck,Stack,IO-Stack)

Theorem [DS12, ATPS]

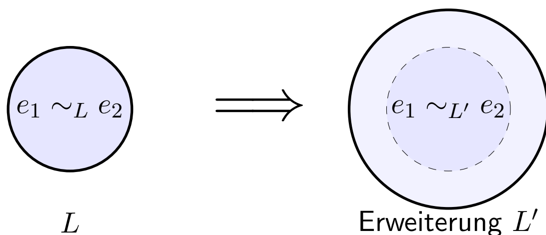
CIOM1 ist ein **korrekter Evaluator** für *CHF*
(Es gilt Konvergenzäquivalenz)

CHF als Erweiterung

Motivation:

- Pures funktionales Haskell gut untersucht (Gleichheiten, Techniken)
- Kann diese Theorie auch auf CHF übertragen werden?

Eine Erweiterung ist **konservativ** wenn sie alle Gleichheiten erhält



CHF ist eine konservative Erweiterung [SSS12,LICS]

PF = Pure, determ. Subsprache von CHF, keine Futures, kein I/O

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

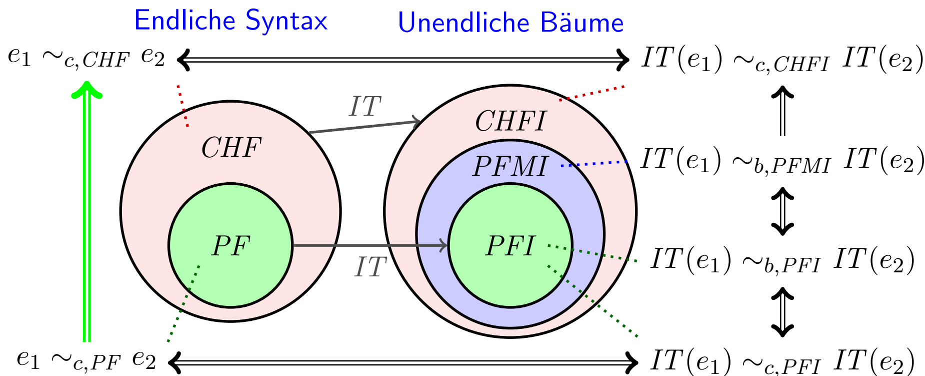
Theorem [SSS12,LICS]

CHF erweitert PF konservativ

$$\forall e_1, e_2 :: \tau \in \mathit{Expr}_{PF}\colon e_1 \sim_{c,PF} e_2 \implies e_1 \sim_{c,CHF} e_2.$$

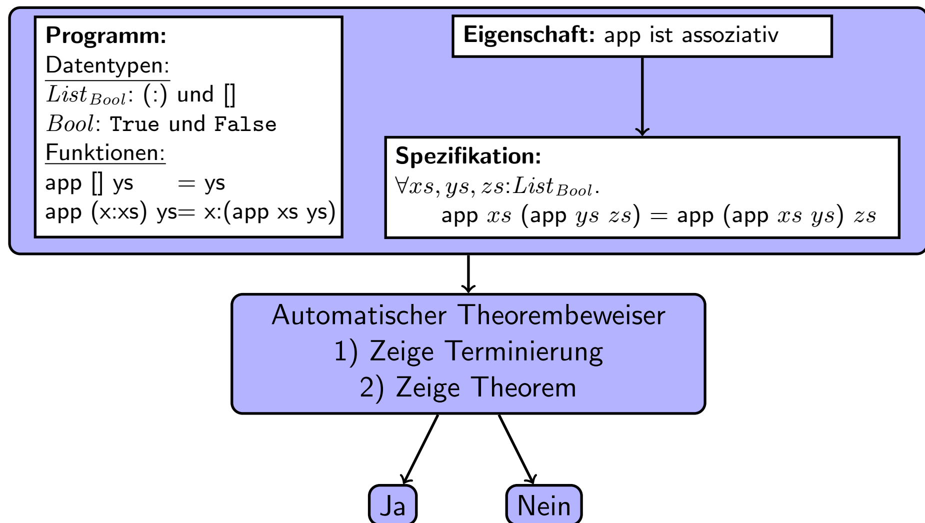
$\Longrightarrow$ korrekte Transformationen in PF gelten auch in CHF

Beweisstruktur

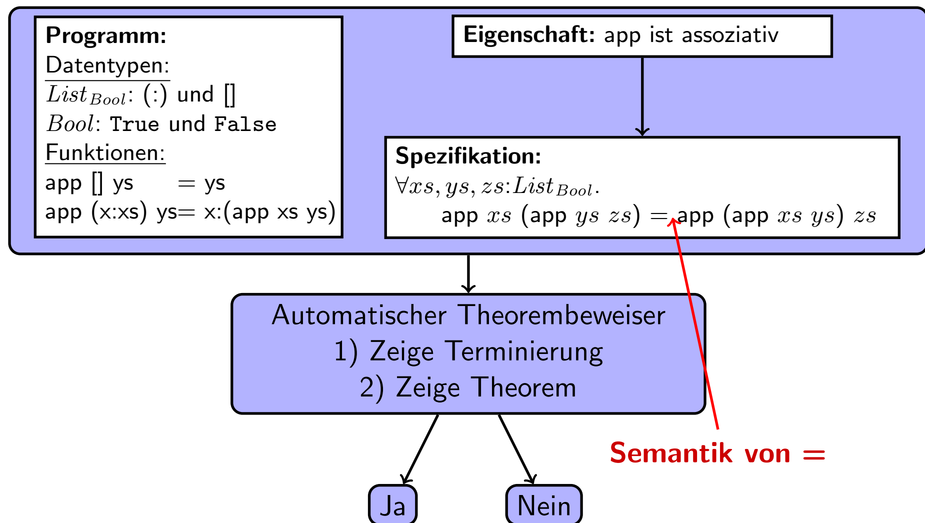


Eine zweiwertige Logik für eine funktionale call-by-value Sprache mit kontextueller Äquivalenz

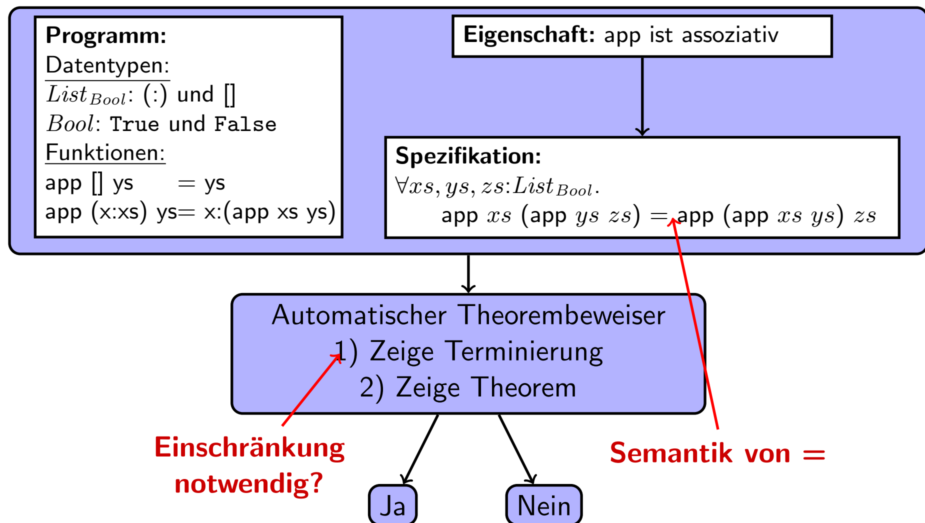
Motivation



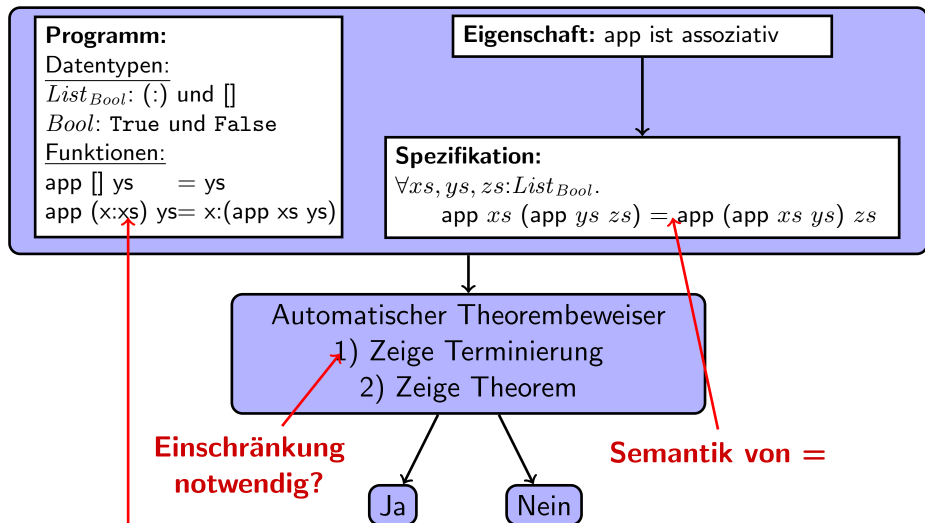
Motivation



Motivation



Motivation



Reicht es aus nur dieses Programm zu betrachten?

Überblick [SSS12,JAR]

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

- Kontextuelle Äquivalenz als Semantik von $=$
- First-Order Logik mit $(e_1 = e_2)$ als Atome (e_1,e_2 Programme)
- Call-by-value higher-order Kernsprache
- Terminierung nicht notwendig
- Tautologische Formeln: Gültigkeit bzgl. aller Programmerweiterungen

Konservativität (1)

Definition

Eine Klasse X von $\mathcal{P}$-Formeln ist konservativ bezüglich Programmerweiterungen gdw.
Gültigkeit bzgl. Programm $\mathcal{P}$ auch die Gültigkeit bzgl. aller Erweiterungen impliziert.

- konservative Formeln lassen sich beweisen ohne Betrachtung von Programmerweiterungen!
- praktische Relevanz
- Nicht alle Formeln sind konservativ (siehe [SSS12,JAR])

Konservativität (2)

Haupttheorem [SSS12,JAR]

$$\sim_{\mathcal{P}} \quad = \quad \sim_{\forall\mathcal{P}}$$

wobei:

$$e_1 \sim_{\mathcal{P}} e_2 \text{ gdw. } \forall\mathcal{P}\text{-Kontexte } C\text{: } C[e_1]{\downarrow} \iff C[e_2]{\downarrow}$$

$$e_1 \sim_{\forall\mathcal{P}} e_2 \text{ gdw. } \forall\mathcal{P}' \sqsupseteq \mathcal{P} : \forall\mathcal{P}'\text{-Kontexte } C\text{: } C[e_1]{\downarrow} \iff C[e_2]{\downarrow}$$

Konservativität (3)

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Theorem [SSS12,JAR]

Die Klassen der AEQ-, DT-, TP- und ATD-Formeln sind konservativ bezüglich Programmerweiterungen.

- AEQ-Formeln (allquantifzierte Gleichungen): $\forall x_1 :: T_1, \ldots, x_n :: T_n.e_1 = e_2$.
- DT-Formeln Quantifizierte Variablen haben keinen Funktionstyp
- TP-Formeln (Terminierungsbeweise): $\forall x_1 :: T_n, \ldots, x_n :: T_n.\neg(e = \bot)$
- ATD-Formeln $F := \forall x_1 :: T_1, \ldots, x_n :: T_n.F' \implies F''$, wobei F' *liefert Terminierungsbedingung für* F'', z.B. $\forall x :: Nat.\neg(x = 0) \implies (div\ x\ x) = (S\ 0)$

Vorteile von LMF

- Nichtterminierende Funktionen sind behandelbar
- Partielle Funktionen sind einfach auszudrücken

  tail [] $= \bot$
  tail (x:xs) $=$ []
- Die Logik ist trotzdem zweiwertig
- $=$ ist semantisch einwandfrei definiert (als $\sim$)

Ausblick:

- Polymorphe Formeln (erste Ergebnisse in [SSS12,JAR])
- Call-by-need?

Fazit

Fazit

- Korrektheit, Vergleich und Eigenschaften von Programmen und Programmiersprachen
- Mithilfe der Formalen Programmiersprachensemantik
- Ergebnisse in verschiedenen Bereichen insbes. für reale Programmiersprachen
- Kontextuelle Äquivalenz als Basis führt zu guten Resultaten