Zum heutigen Thema und zum weiteren Ablauf der LVA

- Functional Reactive Programming
  ...*vorgeschlagen von Benedikt Huber*
  - Programmierung hybrider Systeme
    Intuitiv... Systeme mit stetigen und diskreten Komponenten, Echtzeitsysteme

    Damit angesprochen...
    * Entwurf domänenspezifischer Sprachen
    * Ihre Einbettung in funktionale Programmiersprachen, insbesondere Haskell
    * Programmiersprachliche Konzepte wie Monaden und "Arrows"
    * ...
- Memoization – Funktionen mit Gedächtnis
- Wohldefiniertheits- und Korrektheitsfragen

Memoization – Die zentrale Idee 1(4)

Memoization – Funktionen mit Gedächtnis...

Trick:

- Intern wird jedes Funktionsargument mit einem *Vermerk* (*engl. memo*) versehen, der den zugehörigen Funktionswert enthält, so dieser bekannt (berechnet worden) ist.

  Jede Auswertung der Funktion für ein Argument legt als Seiteneffekt den Funktionswert im Vermerk ab und liefert diesen, wenn er dort schon vermerkt ist, ohne Neuberechnung zurück.

Memoization – Die zentrale Idee 2(4)

Beobachtung...

Das Konzept *gedächtnisbehafter Funktionen*...

- ...ist paradigmenunabhängig und nicht auf funktionale Programmierung beschränkt.
- ...setzt Seiteneffektfreiheit voraus.

Somit...

Funktionale Programmierung...

- ...aufgrund *referentieller Transparenz* funktionaler Programme und der von ihr implizierten Seiteneffektfreiheit besonders prädestiniert!

Memoization – Die zentrale Idee 3(4)

Kanonisches Beispiel...

Ersetzung der notorisch ineffizienten Implementierung zur Berechnung der Fibonacci-Zahlen...

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1)  +  fib (n-2)
```

...durch eine gedächtnisbehaftete Variante:

```
flist = [ fib x | x <- [0 ..] ]
fib 0 = 1
fib 1 = 1
fib n = flist !! (n-1)  +  flist !! (n-2)
```


Memoization – Die zentrale Idee 4(4)

Beachte...

- Im Beispiel der Fibonacci-Zahlen konnte die gedächtnisbehaftete Version mit rein funktionalen (Sprach-)mitteln realisiert werden

Aber...

- Im allgemeinen ist dies nicht immer möglich oder zweckmäßig

....zuvor aber noch einige weitere Beispiele zu Memoization, in der Folge bezeichnet als *Memoration*...

Memoration – Weitere Beispiele 1(3)

Memoration kann...

- unmittelbar profitabel sein

  ```
  flist = [ fib x | x <- [0 ..] ]
  fib 0 = 1
  fib 1 = 1
  fib n = flist !! (n-1)  +  flist !! (n-2)
  ```

  ...wie im Beispiel der Fibonacci-Zahlen: lohnend ab dem ersten Aufruf

- mittelbar profitabel sein

  ```
  flist = [ fak x | x <- [0 ..] ]
  fak 0 = 1
  fak n = n * flist !! (n-1)
  ```

  ...wie im Beispiel der Fakultätsfunktion: lohnend erst bei wiederholten Aufrufen oder bei Einbettung...

Memoration – Weitere Beispiele 2(3)

...in komplexere Anwendungssituationen:

Berechne die Summe der Binomialkoeffizienten

$$\sum_{i=0}^{n} \binom{n}{i}$$

unter Ausnutzung der Beziehungen

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \tag{1}$$

$$\binom{n}{k} = \binom{n}{n-k} \tag{2}$$

Memoration – Weitere Beispiele 3(3)

Memoration bei komplexeren Datenstrukturen...

Bestimme die tiefsten Blätter in einem Baum...

```
data Tree a = Leaf a | Node a (Tree a) (Tree a)

deepest (Leaf x)                            = [x]
deepest (Node x l r) | depth l > depth r   = deepest l
                     | depth l < depth r   = deepest r
                     | otherwise           = deepest l ++ deepest r

depth (Leaf x) = 0
depth (Node x l r) = 1 + max (depth l) (depth r)
```

Obige Lösung ist...

- elegant
- aber ineffizient (quadratisch in der Größe des Baums!) ohne Memoration (der Ergebnisse der Funktion `depth`)

Memoration – Einsatz i.d. Praxis 1(5)

Trotz offensichtlichen Nutzens von Memoration in vielen Situationen macht man die Beobachtung...

- Memoration in der Praxis nicht weit verbreitet

Memoration – Einsatz i.d. Praxis 2(5)

Mögliche Gründe dafür...

- Übergang zu effizienteren Algorithmen, so dass die Notwendigkeit effizienter gedächtnisbehafteter Realisierungen entfällt.

  Am Beispiel der...

  – Fibonacci-Zahlen

```
fibs :: [Integer]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

  – Berechnung der Summe der Binomialkoeffizienten

$$\sum_{i=0}^{n} \binom{n}{i} = 2^n$$

  Mögliche Nachteile...

  – Aufwändiger für den Programmierer
  (...nicht in obigen beiden Beispielen)

  – Möglicher Verlust der "natürlichen" Struktur des Lösungsalgorithmus

Memoration – Einsatz i.d. Praxis 3(5)

Mögliche weitere Gründe...

- Schwierig zu implementieren, wenn die Werte des Argumentbereichs Elemente eines komplexen Datentyps sind (Bäume, Graphen,...)
  – Bei komplexen Datenstrukturen aufwändiger Vergleich, ob ein bestimmtes Argument bereits behandelt wurde und der zugehörige Funktionswert in der Memo-Tabelle vermerkt ist (trade-off Problem zwischen aufwändigem Vergleich und Wiederverwendung des Funktionsresultats)
  – Hash-Techniken zum Erkennen gleicher (Daten-) Strukturen nur schwer einsetzbar
  – Vollständig geteilte Speicherung gleicher Datenstrukturen, um Test auf Strukturgleichheit auf Zeigergleichheit zurückzuführen, ebenfalls sehr aufwändig
  – s.u.

Memoration – Einsatz i.d. Praxis 4(5)

Mögliche weitere Gründe...

- Schwierig oder aufwändig zu implementieren, wenn die Werte des Argumentbereichs Elemente eines komplexen Datentyps sind (Bäume, Graphen,...)
  – s.o.
  – Ausdehnung auf potentiell unendliche Datenstrukturen erfordert weitere Spezialüberlegungen
  – *Lebenszeitproblem*: Potentiell hoher Speicherverbrauch erfordert "Theorie des Verlernens oder Vergessens" (für welche Argumente wird der Funktionswert mehrfach benötigt, noch benötigt, nicht mehr benötigt,...)
  – Komplexe Interaktionen mit anderen Laufzeitsystemroutinen, etwa der Speicherbereinigung (engl. garbage collection)

Memoration – Einsatz i.d. Praxis 5(5)

Mögliche weitere Gründe...

- Zwang zu Paradigmenbrüchen (etwa bei seiteneffektbehafteten Realisierungen in funktionalen Programmierumgebungen; möglicher Verlust referentieller Transparenz,...)
- In verteilten Umgebungen:
  – *Deadlockproblem*: Verteilte Funktionsauswertung erfordert ausgeklügelte Synchronisationsmechanismen beim Zugriff auf die Memo-Tabelle; Gefahr möglicher Verklemmung der beteiligten Prozesse
  – *Kommunikationsengpass*: Beim Zugriff auf Memo-Tabelle über das Kommunikationsnetz
- ...

Memoration – Implementierungsaspekte 1(4)

Implementierungsoptionen...

- Durch Spracherweiterungen, etwa explizites Hinzunehmen von (Speicher-) Identitätstests und Hash-Tabellen

  Aber: I.a. mit Verlust referentieller Transparenz verbunden (vgl. Vorschlag von F.L. Morris, 1981)

Memoration – Implementierungsaspekte 2(4)

Auf Robert Keller und Ronan Sleep geht folgender Vorschlag zurück:

Erreichen von...

- Programmierer in die Lage zu versetzen, durch geeignete Spezifikationen Wiederberechnungen zu vermeiden
- das Ausmaß von Wiederberechnungen selektiv kontrollieren zu lassen
- geeignete Kontrollmechanismen sich dafür zu schreiben

...durch

- zur Verfügungstellung geeigneter applikativer Bausteine an der Schnittstelle zwischen Anwendungsprogrammierung und Übersetzung

Memoration – Implementierungsaspekte 3(4)

Auf John Hughes geht folgender Vorschlag zurück:

- *Lazy Memo-Functions*

  Idee... Ersetzung des Tests auf *strukturelle Gleichheit* durch Test auf *Identität* (Objekte sind gleich, wenn sie am selben Speicherplatz abgelegt sind!)

  Vorteile:

  – Hohe Effizienz: Gleichheitstest auf Zeigergleichheit zurückgeführt
  – Gute Realisierbarkeit: durch Rückgriff auf Identitätstests, Hash-Tabellen, Zuweisungen

  Nachteile:

  – Geringere Generalität: Gleiche Argumente werden nicht immer erkannt und erfordern im Prinzip unnötige Funktionswertwiederberechnungen

Memoration – Implementierungsaspekte 4(4)

Implementierungsoptionen – Generell:

Memoration realisierbar über...

- Memo-Tabellen unter Verwendung destruktiver Zuweisungen
  ...*Bruch des rein funktionalen Programmierparadigmas*
- Memo-Tabellen, die als Argument einer memo-Funktion übergeben werden, die als Ergebnis eine neue Memo-Tabelle liefert
  ...*kein Bruch des funktionalen Programmierparadigmas, aber weniger effizient*

Memoration – Ein kurzes Fazit

Memoration...

- spezielle Programmierungs- bzw. Optimierungstechnik zur Performanzsteigerung durch
  - Anwendungsprogrammierer
  - Übersetzer
- grundsätzlich paradigmenunabhängig, jedoch paradigmensensitiv bezüglich Umsetzbarkeit und Wirksamkeit
- keine "*silver bullet*" zur Performanzsteigerung, sondern wie jede Optimierung in ihrer Einsetzbarkeit und Wirksamkeit von Anwendungssituation und -szenario abhängig

Literaturhinweise zum Thema "Funktionen mit Gedächtnis"

- D. Michie. *"Memo" Functions and Machine Learning*". Nature, 218, 19-22, 1968.
- Anthony J. Field, Peter G. Harrison. *Functional Programming*. Addison-Wesley, 1988, Kapitel 19.
- Robert M. Keller, M. Ronan Sleep. *Applicative Caching: Programmer Control of Object Sharing and Lifetime in Distributed Implementations of Applicative Languages*. In Proceedings of the ACM Conference on Functional Languages and Computer Architecture, Wentworth, NH, 1981, pp. 131-147.
- John Hughes. *Lazy Memo-functions*. In Proceedings of the Conference on Functional Languages and Computer Architecture, LNCS 201, Springer-Verlag, Nancy, France, 1985, pp. 129-146.

Wohldefiniertheits- & Korrektheitsfragen

- *Ströme und Funktionen auf Strömen*
  ...fundiert und wohldefiniert?
- *Korrektheit von Programmen, Beweis von Programmeigenschaften*
  ...Rekursion vs. Induktion, Induktionsbeweise

Dazu zunächst...

- *Mathematische Grundlagen*
  ...CPOs, Fixpunkte, Fixpunktsätze

Literaturhinweise

Die folgende Darstellung orientiert sich u.a. an...

- Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications – A Formal Introduction*. Wiley, 1992.
  `http://www.daimi.au.dk/~bra8130/Wiley_book/wiley.html`
- Kapitel 11 und 14
  Paul Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia*. Cambridge University Press, 2000.
- Kapitel 8 und 17
  Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley, 2nd edition, 1999.

Unendliche Listen, Fixpunkte und Gleichungssysteme

- Ströme
  - `onetwo = 1 : 2 : onetwo`
    ⇝ `[1,2,1,2,1,2,...`
  - `onestwos = 1 : onestwos : 2`
    ⇝ `[1,1,1,1,1,1,...`
- Gleichungssysteme
  - `x = E[x]`

Mehr dazu im folgenden...

Mengen und Relationen 1(2)

Sei M eine Menge und R eine Relation auf M, d.h. $R \subseteq M \times M$.

Dann heißt R...

- *reflexiv* gdw. $\forall m \in M.\ m\,R\,m$
- *transitiv* gdw. $\forall m,n,p \in M.\ m\,R\,n \ \wedge\ n\,R\,p \ \Rightarrow\ m\,R\,p$
- *antisymmetrisch* gdw. $\forall m,n \in M.\ m\,R\,n \wedge n\,R\,m \ \Rightarrow\ m = n$

Darüberhinaus... (wenn auch in der Folge weniger wichtig)

- *symmetrisch* gdw. $\forall m,n \in M.\ m\,R\,n \iff n\,R\,m$
- *total* gdw. $\forall m,n \in M.\ m\,R\,n \ \vee\ n\,R\,m$

Mengen und Relationen 2(2)

Eine Relation R auf M heißt

- *Quasiordnung* gdw. R ist reflexiv und transitiv
- *partielle Ordnung* gdw. R ist reflexiv, transitiv und antisymmetrisch

Zur Vollständigkeit sei ergänzt...

- *Äquivalenzrelation* gdw. R ist reflexiv, transitiv und symmetrisch

...eine partielle Ordnung ist also eine antisymmetrische Quasiordnung, eine Äquivalenzrelation eine symmetrische Quasiordnung.

Schranken, kleinste, größte Elemente

Sei $(Q, \sqsubseteq)$ eine Quasiordnung, sei $q \in Q$ und $Q' \subseteq Q$.

Dann heißt q...

- *obere* (*untere*) *Schranke* von Q', in Zeichen: $Q' \sqsubseteq q$ ($q \sqsubseteq Q'$), wenn für alle $q' \in Q$ gilt: $q' \sqsubseteq q$ ($q \sqsubseteq q'$)
- *kleinste obere* (*größte untere*) *Schranke* von Q', wenn q obere (untere) Schranke von Q' ist und für jede andere obere (untere) Schranke $\hat{q}$ von Q' gilt: $q \sqsubseteq \hat{q}$ ($\hat{q} \sqsubseteq q$)
- *größtes* (*kleinstes*) Element von Q, wenn gilt: $Q \sqsubseteq q$ ($q \sqsubseteq Q$)

Eindeutigkeit von Schranken

- In partiellen Ordnungen sind kleinste obere und größte untere Schranken eindeutig bestimmt, wenn sie existieren.
- Existenz (und damit Eindeutigkeit) vorausgesetzt, wird die kleinste obere (größte untere) Schranke einer Menge $P' \subseteq P$ der Grundmenge einer partiellen Ordnung $(P, \sqsubseteq)$ mit $\bigsqcup P'$ ($\bigsqcap P'$) bezeichnet. Man spricht dann auch vom *Supremum* und *Infimum* von P'.
- Analog für kleinste und größte Elemente. Existenz vorausgesetzt, werden sie üblicherweise mit $\perp$ und $\top$ bezeichnet.

Verbände und vollständige Verbände

Sei $(P, \sqsubseteq)$ eine partielle Ordnung.

Dann heißt $(P, \sqsubseteq)$...

- *Verband*, wenn jede *endliche* Teilmenge P' von P eine kleinste obere und eine größte untere Schranke in P besitzt
- *vollständiger Verband*, wenn *jede* Teilmenge P' von P eine kleinste obere und eine größte untere Schranke in P besitzt

...(vollständige) Verbände sind also spezielle partielle Ordnungen.

Vollständige partielle Ordnungen

...ein etwas schwächerer, aber in der Informatik oft ausreichender und daher angemessenerer Begriff.

Sei $(P, \sqsubseteq)$ eine partielle Ordnung.

Dann heißt $(P, \sqsubseteq)$...

- *vollständig*, kurz *CPO* (von engl. complete partial order), wenn jede aufsteigende Kette $K \subseteq P$ eine kleinste obere Schranke in P besitzt.

Es gilt:

- Eine CPO $(C, \sqsubseteq)$ (genauer wäre: "kettenvollständige partielle Ordnung (engl. chain complete partial order (CCPO)") besitzt stets ein kleinstes Element, eindeutig bestimmt als Supremum der leeren Kette und üblicherweise mit $\perp$ bezeichnet: $\perp =_{df} \bigsqcup \emptyset$.

Ketten

Sei $(P, \sqsubseteq)$ eine partielle Ordnung.

Eine Teilmenge $K \subseteq P$ heißt...

- *Kette* in P, wenn die Elemente in K total geordnet sind. Für $K = \{k_0 \sqsubseteq k_1 \sqsubseteq k_2 \sqsubseteq \ldots\}$ ($\{k_0 \sqsupseteq k_1 \sqsupseteq k_2 \sqsupseteq \ldots\}$) spricht man auch genauer von einer *aufsteigenden* (*absteigenden*) Kette in P.

Eine Kette K heißt...

- *endlich*, wenn K endlich ist, sonst *unendlich*.

Kettenendlichkeit, endliche Elemente

Eine partielle Ordnung $(P, \sqsubseteq)$ heißt

- *kettenendlich* gdw. P enthält keine unendlichen Ketten

Ein Element $p \in P$ heißt

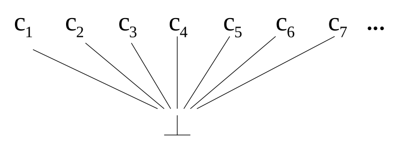
- *endlich* gdw. die Menge $Q =_{df} \{q \in P \mid q \sqsubseteq p\}$ keine unendliche Kette enthält
- *endlich relativ zu* $r \in P$ gdw. die Menge $Q =_{df} \{q \in P \mid r \sqsubseteq q \sqsubseteq p\}$ keine unendliche Kette enthält

(Standard-) CPO-Konstruktionen 1(4)

Flache CPOs...

Sei $(C, \sqsubseteq)$ eine CPO. Dann heißt $(C, \sqsubseteq)$...

- *flach*, wenn für alle $c, d \in C$ gilt: $c \sqsubseteq d \Leftrightarrow c = \perp \ \vee \ c = d$



(Standard-) CPO-Konstruktionen 2(4)

Produktkonstruktionen...

Seien $(P_1, \sqsubseteq_1), (P_2, \sqsubseteq_2), \ldots, (P_n, \sqsubseteq_n)$ CPOs. Dann sind auch...

- das *nichtstrikte* (*direkte*) *Produkt* $(\times P_i, \sqsubseteq)$ mit
  - $(\times P_i, \sqsubseteq) = (P_1 \times P_2 \times \ldots \times P_n, \sqsubseteq)$ mit $\forall\, (p_1, p_2, \ldots, p_n), (q_1, q_2, \ldots, q_n) \in \times P_i.\ (p_1, p_2, \ldots, p_n) \sqsubseteq (q_1, q_2, \ldots, q_n) \Rightarrow \forall\, i \in \{1, \ldots, n\}.\ p_i \sqsubseteq_i q_i$
- und das *strikte* (*direkte*) *Produkt* (*smash Produkt*) mit
  - $(\bigotimes P_i, \sqsubseteq) = (P_1 \otimes P_2 \otimes \ldots \otimes P_n, \sqsubseteq)$, wobei $\sqsubseteq$ wie oben definiert ist, jedoch zusätzlich gesetzt wird:

$$(p_1, p_2, \ldots, p_n) = \perp \Rightarrow \exists\, i \in \{1, \ldots, n\}.\ p_i = \perp_i$$

CPOs.

(Standard-) CPO-Konstruktionen 3(4)

Summenkonstruktion...

Seien $(P_1, \sqsubseteq_1), (P_2, \sqsubseteq_2), \ldots, (P_n, \sqsubseteq_n)$ CPOs. Dann ist auch...

- die *direkte Summe* $(\bigoplus P_i, \sqsubseteq)$ mit...
  - $(\bigoplus P_i, \sqsubseteq) = (P_1 \dot\cup P_2 \;\dot\cup\; \ldots \;\dot\cup\; P_n, \sqsubseteq)$ disjunkte Vereinigung der P_i, $i \in \{1, \ldots, n\}$ und $\forall p, q \in \bigoplus P_i.\ p \sqsubseteq q \Rightarrow \exists i \in \{1, \ldots, n\}.\ p, q \in P_i \ \wedge\ p \sqsubseteq_i q$ und der Identifikation der kleinsten Elemente der $(P_i, \sqsubseteq_i)$, $i \in \{1, \ldots, n\}$, d.h. $\bot =_{df} \bot_i$, $i \in \{1, \ldots, n\}$

  eine CPO.

(Standard-) CPO-Konstruktionen 4(4)

Funktionenraum...

Seien $(C, \sqsubseteq_C)$ und $(D, \sqsubseteq_D)$ zwei CPOs und $[C \to D] =_{df} \{f : C \to D \mid f \text{ stetig}\}$ die Menge der stetigen Funktionen von C nach D.

Dann ist auch...

- der *stetige Funktionenraum* $([C \to D], \sqsubseteq)$ eine CPO mit
  - $\forall f, g \in [C \to D].\ f \sqsubseteq g \iff \forall c \in C.\ f(c) \sqsubseteq_D g(c)$

Funktionen auf CPOs / Eigenschaften

Seien $(C, \sqsubseteq_C)$ und $(D, \sqsubseteq_D)$ zwei CPOs und sei $f : C \to D$ eine Funktion von C nach D.

Dann heißt f...

- *monoton* gdw. $\forall c, c' \in C.\ c \sqsubseteq_C c' \Rightarrow f(c) \sqsubseteq_D f(c')$
  (*Erhalt der Ordnung der Elemente*)
- *stetig* gdw. $\forall C' \subseteq C.\ f(\bigsqcup_C C') =_D \bigsqcup_D f(C')$
  (*Erhalt der kleinsten oberen Schranken*)

Sei $(C, \sqsubseteq)$ eine CPO und sei $f : C \to C$ eine Funktion auf C.

Dann heißt f...

- *inflationär* (*vergrößernd*) gdw. $\forall c \in C.\ c \sqsubseteq f(c)$

Funktionen auf CPOs / Resultate

Mit den vorigen Bezeichnungen gilt...

Lemma
f ist monoton gdw. $\forall C' \subseteq C.\ f(\bigsqcup_C C') \sqsupseteq_D \bigsqcup_D f(C')$

Korollar
Eine stetige Funktion ist stets monoton, d.h. f stetig $\Rightarrow$ f monoton.

(Kleinste und größte) Fixpunkte 1(2)

Sei $(C, \sqsubseteq)$ eine CPO, $f : C \to C$ eine Funktion auf C und sei c ein Element von C, also $c \in C$.

Dann heißt c ...

- *Fixpunkt* von f gdw. $f(c) = c$

Ein Fixpunkt c von f heißt...

- *kleinster Fixpunkt* von f gdw. $\forall d \in C.\ f(d) = d \Rightarrow c \sqsubseteq d$
- *größter Fixpunkt* von f gdw. $\forall d \in C.\ f(d) = d \Rightarrow d \sqsubseteq c$

(Kleinste und größte) Fixpunkte 2(2)

Seien $d, c_d \in C$. Dann heißt c_d ...

- *bedingter kleinster Fixpunkt* von f bezüglich d gdw. c_d ist der kleinste Fixpunkt von C mit $d \sqsubseteq c_d$, d.h. für alle anderen Fixpunkte x von f mit $d \sqsubseteq x$ gilt: $c_d \sqsubseteq x$.

Bezeichnungen:
Der kleinste bzw. größte Fixpunkt einer Funktion f wird oft mit μf bzw. νf bezeichnet.

Fixpunktsatz

Theorem (Knaster/Tarski, Kleene)
Sei $(C, \sqsubseteq)$ eine CPO und sei $f : C \to C$ eine stetige Funktion auf C.

Dann hat f einen kleinsten Fixpunkt μf und dieser Fixpunkt ergibt sich als kleinste obere Schranke der Kette (sog. *Kleene-Kette*) $\{\bot, f(\bot), f^2(\bot), \ldots\}$, d.h.

$$\mu f = \bigsqcup_{i \in \mathbb{N}_0} f^i(\bot) = \bigsqcup\{\bot, f(\bot), f^2(\bot), \ldots\}$$

Beweis des Fixpunktsatzes 1(4)

Zu zeigen: μf...

1. existiert
2. ist Fixpunkt
3. ist kleinster Fixpunkt

Beweis des Fixpunktsatzes 2(4)

1. *Existenz*
   - Es gilt $f^0 \perp = \perp$ und $\perp \sqsubseteq c$ für alle $c \in C$.
   - Durch vollständige Induktion lässt sich damit zeigen: $f^n \perp \sqsubseteq f^n c$ für alle $c \in C$.
   - Somit gilt $f^n \perp \sqsubseteq f^m \perp$ für alle n, m mit $n \leq m$. Somit ist $\{f^n \perp \mid n \geq 0\}$ eine (nichtleere) Kette in C.
   - Damit folgt die Existenz von $\bigsqcup_{i \in \mathbb{N}_0} f^i(\perp)$ aus der CPO-Eigenschaft von $(C, \sqsubseteq)$.

Beweis des Fixpunktsatzes 3(4)

2. *Fixpunkteigenschaft*

$$\begin{array}{rcl} & & f(\bigsqcup_{i \in \mathbb{N}_0} f^i(\perp)) \\ (f \text{ stetig}) & = & \bigsqcup_{i \in \mathbb{N}_0} f(f^n \perp) \\ & = & \bigsqcup_{i \in \mathbb{N}_1} f^n \perp \\ (K \text{ Kette} \Rightarrow \bigsqcup K = \perp \sqcup \bigsqcup K) & = & \bigsqcup_{i \in \mathbb{N}_1} f^n \perp \ \sqcup \ \perp \\ (f^0 = \perp) & = & \bigsqcup_{i \in \mathbb{N}_0} f^n \perp \\ & = & \bigsqcup_{i \in \mathbb{N}_0} f^i(\perp) \end{array}$$

Beweis des Fixpunktsatzes 4(4)

3. *Kleinster Fixpunkt*
   - Sei c beliebig gewählter Fixpunkt von f. Dann gilt $\perp \sqsubseteq c$ und somit auch $f^n \perp \sqsubseteq f^n c$ für alle $n \geq 0$.
   - Folglich gilt $f^n \perp \sqsubseteq c$ wg. der Wahl von c als Fixpunkt von f.
   - Somit gilt auch, dass c eine obere Schranke von $\{f^i(\perp) \mid i \in \mathbb{N}_0\}$ ist.
   - Da $\bigsqcup_{i \in \mathbb{N}_0} f^i(\perp)$ nach Definition die kleinste obere Schranke dieser Kette ist, gilt wie gewünscht $\bigsqcup_{i \in \mathbb{N}_0} f^i(\perp) \sqsubseteq c$.

Bedingte Fixpunkte

Theorem (Endliche Fixpunkte)
Sei $(C, \sqsubseteq)$ eine CPO, sei $f : C \to C$ eine stetige, inflationäre Funktion auf C und sei $d \in C$.

Dann hat f einen kleinsten bedingten Fixpunkt μf_d und dieser Fixpunkt ergibt sich als kleinste obere Schranke der Kette $\{d, f(d), f^2(d), \ldots\}$, d.h.

$$\mu f_d = \bigsqcup_{i \in \mathbb{N}_0} f^i(d) = \bigsqcup \{d, f(d), f^2(d), \ldots\}$$

Endliche Fixpunkte

Theorem (Endliche Fixpunkte)
Sei $(C, \sqsubseteq)$ eine CPO und sei $f : C \to C$ eine stetige Funktion auf C.

Dann gilt: Sind in der Kleene-Kette von f zwei aufeinanderfolgende Glieder gleich, etwa $f^i(\perp) = f^{i+1}(\perp)$, so gilt $\mu f = f^i(\perp)$.

Existenz endlicher Fixpunkte

Hinreichende Bedingungen für die Existenz endlicher Fixpunkte sind...

- Endlichkeit von Definitions- und Wertebereich von f
- f ist von der Form $f(c) = c \sqcup g(c)$ für monotones g über kettenendlichem Wertebereich

Kegel und Ideale

Sei $(P, \sqsubseteq)$ eine partielle Ordnung und Q eine Teilmenge von P, also $Q \subseteq P$.

Dann heißt Q ...

- *gerichtet* (*gerichtete Menge*) (engl. directed set), wenn jede *endliche* Teilmenge $R \subseteq Q$ ein Supremum in Q besitzt, d.h. $\exists q \in Q.\ q = \bigsqcup R$
- *Kegel*, wenn Q nach unten abgeschlossen ist, d.h. $\forall q \in Q\ \forall p \in P.\ p \sqsubseteq q \Rightarrow p \in Q$
- *Ideal*, wenn Q ein gerichteter Kegel ist, d.h. wenn Q nach unten abgeschlossen ist und jede endliche Teilmenge ein Supremum in Q besitzt.

Beachte: Für gerichtete Mengen Q gilt wg. $\emptyset \subseteq Q$ und somit $\bigsqcup \emptyset = \perp \in Q$ stets $Q \neq \emptyset$.

Idealvervollständigung

Theorem (Idealvervollständigung)
Sei $(P, \sqsubseteq)$ eine partielle Ordnung und sei I_P die Menge aller Ideale von P. Dann gilt:

- $(I_P, \subseteq)$ ist eine CPO.

Induzierte "Vervollständigung"...

- Identifikation jedes Element $p \in P$ mit dem zugehörigen Ideal $I_p =_{df} \{q \mid q \sqsubseteq p\}$ liefert eine Einbettung von P in I_P mit $p \sqsubseteq q \Leftrightarrow I_P \subseteq I_Q$

Korollar (Erweiterungskorollar für Funktionen)
Sei $(P, \sqsubseteq_P)$ eine partielle Ordnung und sei $(C, \sqsubseteq_C)$ eine CPO. Dann gilt: Alle monotonen Funktionen $f : P \to C$ lassen sich eindeutig zu stetigen Funktionen $\hat{f} : I_P \to C$ erweitern.

Schlussfolgerung

...voriges Ergebnis impliziert...

- Ströme bilden eine CPO
- Rekursive Gleichungen und Funktionen über Strömen sind wohldefiniert
- Anwendung einer Funktion auf die endlichen Präfixe eines Stroms liefert Kette von Approximationen der Anwendung der Funktion auf den Strom selbst und ist damit korrekt

Korrektheit von Programmen/Beweis von Programmeigenschaften

Beim nächsten Mal...

Vorschau auf die nächsten Vorlesungstermine...

- Di, 10.05.2005, 14 - 16 Uhr, CompLang-Bibliothek
- Di, 24.05.2005, 14 - 16 Uhr, CompLang-Bibliothek