Effizienz der Rekursion

- Wir wollen die Effizienz rekursiver Funktionsdefinitionen studieren.
- Eine bestimmte Art der Rekursion (“*repetitive Rekursion*”) ist besonders platzeffizient.
- Es ist manchmal möglich, rekursive Funktionen in repetitiv rekursive Form zu bringen
- Es gibt Formen der Rekursion, für die solch eine Transformation nur unter Aufbietung zusätzlichen Hilfsspeichers in derselben Größe wie die resultierende Ersparnis möglich ist.

Rekursion im Rechner

Die Auswertung rekursiver Funktionen erfolgt im Rechner im Prinzip so wie in der operationellen Semantik.

In Abwesenheit lokaler und höherstufiger Funktionen können wir näherungsweise die sukzessive Ersetzung von Funktionen durch ihre Definition verwenden ("*Substitutionsmodell*")

Auswertung der Paritätsfunktion

$$\mathit{gerade}(n) = \textbf{if } n = 0 \textbf{ then true else not}(\mathit{gerade}(n-1))$$

$$\mathit{gerade}(4) =$$
$$\textbf{not}(\mathit{gerade}(3)) =$$
$$\textbf{not}(\textbf{not}(\mathit{gerade}(2))) =$$
$$\textbf{not}(\textbf{not}(\textbf{not}(\mathit{gerade}(1)))) =$$
$$\textbf{not}(\textbf{not}(\textbf{not}(\textbf{not}(\mathit{gerade}(0))))) = \textbf{true}$$

Platzverbrauch von $\mathit{gerade}(n)$ ist $O(n)$ aber nicht $O(1)$.

Repetitiv rekursive Version der Parität

$$\mathit{gerade2}(n) = \mathit{gerade2\_aux}(n, \textbf{true})$$

$$\mathit{gerade2\_aux}(n, a) = \textbf{if } n = 0 \textbf{ then } a \textbf{ else } \mathit{gerade2\_aux}(n-1, \textbf{not}(a))$$

$$\mathit{gerade2}(4) = \mathit{gerade2\_aux}(4, \textbf{true}) =$$

$$\mathit{gerade2\_aux}(3, \textbf{not}(\textbf{true})) =$$

$$\mathit{gerade2\_aux}(3, \textbf{false}) =$$

$$\mathit{gerade2\_aux}(2, \textbf{true}) =$$

$$\mathit{gerade2\_aux}(1, \textbf{false}) =$$

$$\mathit{gerade2\_aux}(0, \textbf{true}) = \textbf{true}$$

Platzverbrauch von $\mathit{gerade2}(n) = O(1)$.

Zusammenhang der beiden Funktionen

$$\mathit{gerade2\_aux}(n, a) = \textbf{if } a \textbf{ then } \mathit{gerade}(n) \textbf{ else not}(\mathit{gerade}(n))$$

Beweis durch Satz von der partiellen Korrektheit + Abstiegsfunktion.

Beispiel: Fakultät

$$\mathit{fakt}(n) = \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \cdot \mathit{fakt}(n-1)$$

$$\begin{aligned}
&\mathit{fakt}(4) = \\
&4 \cdot \mathit{fakt}(3)) = \\
&4 \cdot (3 \cdot \mathit{fakt}(2)) = \\
&4 \cdot (3 \cdot (2 \cdot \mathit{fakt}(1))) = \\
&4 \cdot (3 \cdot (2 \cdot (1 \cdot \mathit{fakt}(0)))) = \\
&4 \cdot (3 \cdot (2 \cdot (1 \cdot 1))) = 24
\end{aligned}$$

Der “Computer” kann nicht wissen, dass man die Multiplikationen auch umklammern und vereinfachen kann: die gesamte Multiplikationsaufgabe bleibt stehen, bis endlich $\mathit{fakt}(0)$ ausgewertet wird.

$\rightarrow$ Platzverbrauch von $\mathit{fakt}(n)$ ist $O(n)$ aber nicht $O(1)$.

Repetitiv rekursive Version der Fakultät

$$\mathit{fakt2}(n) = \mathit{fakt2\_aux}(n, 1)$$

$$\mathit{fakt2\_aux}(n, a) = \textbf{if } n = 0 \textbf{ then } a \textbf{ else } \mathit{fakt2\_aux}(n - 1, a \cdot n))$$

$$\mathit{fakt2}(4) = \mathit{fakt2\_aux}(4, 1) =$$

$$\mathit{fakt2\_aux}(3, 1 \cdot 4) =$$

$$\mathit{fakt2\_aux}(3, 4) =$$

$$\mathit{fakt2\_aux}(2, 12) =$$

$$\mathit{fakt2\_aux}(1, 24) =$$

$$\mathit{fakt2\_aux}(0, 24) = 24$$

Platzverbrauch hier $O(1)$.

Zusammenhang der beiden Funktionen

$$\mathit{fakt\_aux}(n, a) = a \cdot \mathit{fakt}(n)$$

Beweis durch Satz von der partiellen Korrektheit + Abstiegsfunktion.

Repetitive Rekursion: Definition

Eine rekursive Definition

$$f(x) = \Phi(f, x)$$

heißt *repetitiv rekursiv* (auch *endständig rekursiv, iterativ rekursiv, endrekursiv, tail recursive*), wenn im Rumpf Φ höchstens ein rekursiver Aufruf von f getätigt wird und dessen Ergebnis dann bereits den Wert von $\Phi(f, x)$ liefert.

Merke: Bei der Auswertung repetitiv rekursiver Funktionen entsteht kein zusätzlicher Platzbedarf für die Verwaltung der Rekursion.

Lineare Rekursion

Eine rekursive Definition heißt *linear*, wenn im Rumpf der Definition höchstens ein rekursiver Aufruf anfällt.

Sehr häufig lassen sich linear rekursive Definitionen durch *Einbettung* in repetitiv rekursive Form bringen.

Dies resultiert dann in beträchtlicher Platz- und Zeitersparnis.

Beispiel: Fakultät, Parität.

Repetitive Rekursion und Iteration

Eine repetitiv rekursive Funktionsdefinition der Form

$$f(x) = \textbf{if } p(x) \textbf{ then } a(x) \textbf{ else } f(b(x))$$

(p beliebiges Prädikat, a, b beliebige Funktionen) kann man auch "imperativ" wie folgt auswerten:

Schreibe die Eingabe in die Speicherstelle x

Solange p auf den Inhalt von x nicht zutrifft, wiederhole folgendes:

Bestimme $y := b(\text{Inhalt von } x)$

Ersetze den Inhalt von x durch y

Gib als Ergebnis $a(\text{Inhalt von } x)$ zurück.

Dasselbe in C

Imperative Programmiersprachen unterstützen diesen Stil, z.B. “C”:

```
f(x) {
   while(!p(x)) {
      x = b(x);
   }
   return a(x);
}
```

Vor- und Nachteile des imperativen Stils

- Imperativer Stil ist zunächst verständlicher
- Funktionale Definition lassen sich durch Gleichungen spezifizieren und verifizieren
- Im imperativen Stil muss man über Zustand der Variablen zu bestimmten Zeitpunkten sprechen, was komplizierter ist.
- In der Effizienz unterscheidet sich die imperative Version nicht von der repetitiv rekursiven Version.

Nichtlineare Rekursion

Sei $f(x) = \Phi(f, x)$ eine rekursive Definition mit Abstiegsfunktion m.

Finden im Rumpf Φ bis zu a rekursive Aufrufe statt (Etwa Fibonacci, Mergesort, Hanoi: $a = 2$), so erzeugt die vollständige Auswertung von $f(x)$ bis zu $a^{m(x)}$ rekursive Aufrufe.

Fibonacci, Hanoi: $m(x) = x \rightarrow$ bis zu 2^x rekursive Aufrufe

Mergesort: $m(x) = O(\log(\mathit{length}(x))) \rightarrow O(\mathit{length}(x)$ rekursive Aufrufe.

Fazit: Bei nichtlinearer Rekursion Laufzeit im Auge behalten.

Fibonacci als repetitive Rekursion

Manchmal lassen sich auch nichtlineare Rekursionen als repetitive R. umschreiben:

$$\mathit{fib}(n) = \textbf{if } n \leq 1 \textbf{ then } 1 \textbf{ else } \mathit{fib}(n-1) + \mathit{fib}(n-2)$$

Definiere

$$\mathit{fib\_aux}(n, a, b) = \textbf{if } n = 0 \textbf{ then } a \textbf{ else } \mathit{fib\_aux}(n-1, b, a+b)$$

Es gilt $\mathit{fib\_aux}(n, \mathit{fib}(k), \mathit{fib}(k+1)) = \mathit{fib}(n+k)$

also $\mathit{fib}(n) = \mathit{fib\_aux}(n, 1, 1)$.

Dies ist ein Spezialfall der *dynamischen Programmierung*, ein allgemeines Optimierungsschema für nichtlineare Rekursionen $\rightarrow$ 'VL "Effiziente Algorithmen".

Echte nichtlineare Rekursion

Bei Mergesort, Linearisierung von Bäumen, Hanoi, lässt sich die nichtlineare Rekursion nur “künstlich” vermeiden, indem man Hilfsdatenstrukturen einführt, z.B. Stapel, die genauso viel Platz verbrauchen, wie die Verwaltung der ursprünglichen Rekursion.

Hier ist also keine echte Verbesserung möglich. In diesen Fällen ist die nichtlineare Rekursion eine vernünftige Lösung.

Grundsätzlich ist im Frühstadium der Softwareentwicklung eine klare rekursive Lösung vorzuziehen.

Bei modularer Planung kann diese später ggf. durch eine effizientere Lösung ersetzt werden.

Erinnerung

Eine rekursive Definition kann auch deshalb ineffizient sein, weil jeder einzelne Verarbeitungsschritt aufwendig ist, nicht weil die Zahl der rekursiven Aufrufe selbst zu groß wäre.

Beispiele: Sortieren durch Einfügen, ineffiziente Version des Umdrehens einer Liste mit “append” (Folie 221).

Hier haben wir jeweils $O(n)$ rekursive Aufrufe, deren jeder aber Aufwand $O(n)$ verursacht, also Gesamtaufwand $O(n^2)$.

Effiziente Datenstrukturen

Oft liegt der Schlüssel zu einem effizienten Alorithmus in der Wahl de geeigneten Datenstruktur.

Als Beispiel betrachten wir nochmal die Binären Suchbäume (BST), Folie 245.

Wir haben gesehen, dass das Suchen eines Elementes im BST t Zeit $O(\text{Höhe}(t))$ erfordert.

Einfügen und Löschen in BST

Man schreibe rekursive OCAML-Funktionen

```
insert : 'a -> 'a bintree -> 'a bintree
delete : 'a -> 'a bintree -> 'a bintree
```

die ein Element in einen BST einfügen, bzw. entfernen.

Einfügen

```
let rec insert x t = match t with
  Empty -> Build(x,Empty,Empty)
| Build(a,l,r) -> if x<=a then Build(a,insert x l,r)
                          else Build(a,l,insert x r)
```

Entfernen

```
let rec delete x t = match t with
  Build(a,l,r) -> if x<a then Build(a,delete x l,r)
                  else if x>a then Build(a,l,delete x r)
                  else if l=Empty then r
                  else if r=Empty then l
                  else let m,l' = remove_max l in
                  Build(m,l',r)
```

Hilfsfunktion `remove_max`

```
let rec remove_max t = match t with
  Build(a,l,Empty) -> a,l
| Build(a,l,r) -> let m,r' = remove_max r in
                  (m,Build(a,l,r'))
```

Knotenanzahl und Höhe

Sei n die Knotenanzahl eines Baumes t und h seine Höhe.

Im besten Fall gilt $n = 2^h - 1$, also $h = O(\log n)$.

Im schlechtesten Fall gilt $n = h$, also $h = O(n)$.

Suchen, Einfügen, Löschen haben Komplexität $O(h)$, also $O(\log n)$ bzw. $O(n)$ je nach Art von t.

Zwei extreme Beispiele

```
let t1 = insert 6 (insert 5 (insert 4 (insert 3
         (insert 2 (insert 1 (insert 0 Empty))))))
let t2= insert 6 (insert 4 (insert 1 (insert 0
         (insert 5(insert 1 (insert 3 Empty))))))
```

`t1` hat maximale Höhe (6) und `t2` hat minimale Höhe (3)

Eine Menge von Bäumen $\mathcal{B}$ heißt *balanciert*, wenn

$$\max\{\text{Höhe}(t) \mid \texttt{knotanz}(t) \leq n, t \in \mathcal{B}\} = O(\log(n))$$

Für balancierte BST mit n Knoten erfordern Einfügen, Löschen, Suchen Zeit $O(\log(n))$.

Rotationen

Man kann Bäume balanciert halten durch systematisches Anwenden der folgenden beiden Rotationen auf Teilbäume Verlauf des Einfügens und Löschens.

```
(* rotate_left : 'a bintree -> 'a bintree *)
let rotate_left = function
    Build(a,l,Build(b,m,r)) -> Build(b,Build(a,l,m),r)
(* rotate_right : 'a bintree -> 'a bintree *)
let rotate_right = function
    Build(a,Build(b,l,m),r) -> Build(b,l,Build(a,m,r))
```

Übung: man begründe, dass die Rotationen die BST-Eigenschaft erhalten.

Um effizient feststellen zu können, welche Rotation angewendet werden soll, muss man in den Bäumen zusätzliche Information speichern, z.B.: Differenz der Knotenanzahl zwischen linkem und rechten Teilbäumen. Näheres: Info II.

Kapitel 7: Semantik und Fixpunkttheorie

Operationelle Semantik von Listen und Bäumen

Sei $\mathcal{L}$ eine *unendliche* Menge von *Adressen* (im Speicher). Z.B.: $\mathcal{L} = \mathbb{N}$.

Die *Werte* i.S.d. operationellen Semantik (vgl. Folie 157) werden erweitert um

- Jede Adresse $\ell \in \mathcal{L}$ ist ein Wert.

Halde

Eine *Halde* (engl.: *heap*)[a] ist eine endliche Funktion von Adressen ($\mathcal{L}$) auf Werte.

Ist h eine Halde, so bezeichnet $\mathrm{dom}(h)$ die (endliche) Teilmenge von $\mathcal{L}$, für die h definiert ist.

$h[\ell \mapsto v]$ ist wie üblich durch

$$h[\ell \mapsto v](\ell') = \begin{cases} v, \text{ wenn } \ell = \ell' \\ h(\ell'), \text{ sonst} \end{cases}$$

definiert. Beachte: $\mathrm{dom}(h[\ell \mapsto v] = \mathrm{dom}(h) \cup \{\ell\}$.

[a] Halde (heap) ist ein "Teekesselchen": Man verwendet den Begriff auch für eine bestimmte baumartige Datenstruktur; das hat mit der Verwendung hier *nichts* zu tun.

Operationelle Semantik mit Halde

Zur Modellierung dynamischer Datenstrukturen erweitern wir das *Format* der operationellen Semantik um Halden wie folgt:

$$U, h, e \to w, h'$$

bedeutet, dass in der Umgebung U und Halde h der Ausdruck e den Wert w hat. Zudem verändert sich durch die Auswertung die Halde zu h'.

Wie zuvor wird dieses *Urteil* durch Auswerteregeln definiert:

Auswerteregeln

- ist c eine Konstante, so gilt $U, h, c \to c, h$. (Halde verändert sich nicht.)
- ist x ein Bezeichner und $<x, w> \in U$, so gilt $U, x, h \to w, h$.
- für Funktionsausdrücke gilt $U, h,$ `fun` x`->`$e \to ($`fun` x`->`$e, U'), h$, wobei U' die Einschränkung von U auf die freien Variablen von e ist.

Auswerteregeln

- ist *op* ein Infixoperator aber nicht `||`, `&&`, welcher eine Basisfunktion $\oplus$ bezeichnet so gilt folgendes: $U, h, x \mathit{op} y \to U(x) \oplus U(y), h$.

  Anwendung von *op* auf geschachtelte Ausdrücke kann man durch Einfügen von `lets` auf diesen Fall reduzieren.

Einstellige Basisfunktionen werden analog behandelt.

Bindung

- Um w, h' mit $U, h,$ `let` x`=`e_1 `in` $e_2 \to w, h'$ zu bestimmen, muss zunächst w_1, h_1 mit $U, h, e_1 \to w_1, h_1$ ermittelt werden. Sodann sind w_2, h_2 mit $U + \{<x, w_1>\}, h_1, e_2 \to w_2, h_2$ zu bestimmen. Es ist dann $U, h,$ `let` x`=`e_1 `in` $e_2 \to w_2, h_2$.

Konstruktoren

- $U, h, [] \to \ell, h[\ell \mapsto (0,0,0)]$ wobei $\ell \notin \mathrm{dom}(h)$.
- $U, h, x::y \to \ell, h[\ell \mapsto (1, U(x), U(y))$, wobei $\ell \notin \mathrm{dom}(h)$

Konstruktoren anderer (rekursiver) Varianten werden analog behandelt.

Mustervergleich (einfachster Fall)

Sei $e = \texttt{match}\ \ x\ \ \texttt{with []->}e_1\texttt{|}y\texttt{::}z\texttt{->}e_2$.

Um h', w mit $U, h, e \to w, h'$ zu bestimmen, gehe man wie folgt vor:

- Falls $U(z) = \ell$ und $h(\ell) = (0, 0, 0)$, so bestimme man w_1, h_1 mit $U, h, e_1 \to w_1, h_1$. Es ist dann $U, h, e \to w_1, h_1$.
- Falls $U(z) = \ell$ und $h(\ell) = (1, u, v)$, so bestimme man w_2, h_2 mit $U[y{\mapsto}u, z{\mapsto}v], h, e_2 \to w_2, h_2$. Es ist dann $U, h, e \to w_2, h_2$.

Beispiel

Sei $e_1 =$ `7::3::10::[]`.

Es gilt $\emptyset, h, e_1 \to \ell_4, h_1$, wobei

$$h_1 = h[\ell_4 \mapsto (1, 7, \ell_3), \ell_3 \mapsto (1, 3, \ell_2), \ell_2 \mapsto (1, 10, \ell_1), \ell_1 \mapsto (0, 0, 0)]$$

und $\ell_1, \ell_2, \ell_3, \ell_4$ paarw. verschieden und nicht in $\mathrm{dom}(h)$.

Sei $e_2 =$ `9::123::[]`. Es gilt $\emptyset, h_1, e_1 \to \ell_7, h_2$, wobei

$$h_2 = h_1[\ell_7 \mapsto (1, 9, \ell_6), \ell_6 \mapsto (1, 123, \ell_5), \ell_5 \mapsto (0, 0, 0)$$

und ℓ_5, ℓ_6, ℓ_7 paarw. verschieden und nicht in
$\mathrm{dom}(h_1) = \mathrm{dom}(h) \cup \{\ell_1, \ell_2, \ell_3\}$.

Beispiel

Es sei `append` wie üblich definiert:

```
let rec append l1 l2 = match l1 with []->l2|y::z->y::append z l2
```

Es sei $U = \emptyset[\texttt{l1} \mapsto \ell_3, \texttt{l2} \mapsto \ell_7]$.

Es gilt $U, h \to$ `append l1 l2` $\to \ell_{10}, h_3$, wobei

$$h_3 = h_2[\ell_{10} \mapsto (1, 7, \ell_9), \ell_9 \mapsto (1, 3, \ell_8), \ell_8 \mapsto (1, 10, \ell_7)]$$

Beispiel

Sei $e =$

```
let l1 = 7::3::10::[] in
let l2 = 9::123::[] in
append l1 l2
```

Es gilt

$$\emptyset, h, e \to \ell_{10} h_3$$

Die Adressen $\ell_1, \ell_2, \ell_3, \ell_4$ sind in $\mathrm{dom}(h_3)$, aber ihre Inhalte sind nicht mehr zugreifbar. Es wäre wünschenswert, zu setzen

$$\emptyset, h, e \to \ell_{10}, h_4$$

wobei $\mathrm{dom}(h_4) = \mathrm{dom}(h_3) \setminus \{\ell_1, \ell_2, \ell_3, \ell_4\}$ und $h_4(\ell) = h_3(\ell)$ für alle $\ell \in \mathrm{dom}(h_4)$.

Garbage collection

In der operationellen Semantik wird die Halde immer größer und enthält immer mehr unerreichbare Adressen.

In praktischen Implementierungen werden unerreichbare Adressen regelmäßig aus der Halde entfernt.

Diesen Prozess bezeichnet man als *garbage collection* (Müllabfuhr).

Dazu werden ausgehend von der aktuellen Umgebung alle erreichbaren Adressen markiert und anschließend alle nichtmarkierten Adressen gelöscht.

Konkrete Implementierung und Formalisierung dieses Prozesses: Spezialvorlesung im Hauptstudium.

Garbage collection gibt es in funktionalen und manchen objektorientierten Programmiersprachen, z.B.: Java.

Denotationelle Semantik, genauer

Die denotationelle Semantik wie in Kap 3.4, weist noch einige Ungenauigkeiten auf:

- Was sind überhaupt “Werte”?
- Was bedeutet “. . . die durch . . . definierte rekursive Funktioni. . . ”?
- Warum “gibt” es rekursiv definierte Funktionen?
- Wie harmoniert Rekursion mit Funktionen höherer Ordnung?

Beispiel

```
let rec fix phi = fun x -> phi (fix phi) x
(* fix :  ((’a -> ’b) -> ’a -> ’b) -> ’a -> ’b*)
let fakt = fix (fun f x -> if x=0 then 1 else x*f(x-1))
(* fakt : int -> int *)
```

Hier liefert `fakt 3` das Ergebnis 6, wie erwartet.

Definiert man aber

```
let rec fix phi = phi (fix phi)
```

so terminiert schon

```
let fakt = fix (fun f x -> if x=0 then 1 else x*f(x-1))
```

nicht.

Durch die denotationelle Semantik wird dies nicht zufriedenstellend erklärt.

Partielle Ordnung

Eine binäre Relation $\sqsubseteq$ auf einer Menge A heißt *partielle Ordnung*, wenn für alle $x, y, z \in A$ folgendes gilt:

- $x \sqsubseteq x$ (Reflexivität)
- $x \sqsubseteq y, y \sqsubseteq x \rightarrow x = y$ (Antisymmetrie)
- $x \sqsubseteq y, y \sqsubseteq z \rightarrow x \sqsubseteq z$ (Transitivität)

Beispiele

- $A = \mathbb{N}, x \sqsubseteq y \leftrightarrow x \leq y$
- $A = \mathbb{N}, x \sqsubseteq y \leftrightarrow x \geq y$
- A beliebig, $x \sqsubseteq y \leftrightarrow x = y$ (dies nennt man die *diskrete*[a] *Ordnung*)
- $A = \mathfrak{P}(X), x \sqsubseteq y \leftrightarrow x \subseteq y$
- $A = X \cup \{\bot\}, x \sqsubseteq y \leftrightarrow x = \bot \vee x = y$

Es ist üblich, die partielle Ordnung durch die *Trägermenge* zu bezeichnen und immer das Symbol $\sqsubseteq$ für die Relation zu verwenden.

[a] Im Englischen gibt es *discrete* und *discreet*. Hier meinen wir ersteres.

Begriffe

Seien X, Y partielle Ordnungen. Alle betrachteten Funktionen seien total.

- Eine Folge $(x_i)_{i\in\mathbb{N}}$ von Elementen $x_i \in X$ heißt *ω-Kette*, kurz *Kette*, wenn $x_i \sqsubseteq x_{i+1}$ für alle i.
- $f : X \to Y$ heißt *monoton*, wenn $x \sqsubseteq x'$ impliziert $f(x) \sqsubseteq f(x')$.
- Sei $U \subseteq X$. Ein Element $x \in X$ heißt *kleinste obere Schranke* oder *Supremum* von U, wenn gilt
  - $u \sqsubseteq x$ für alle $u \in U$,
  - Falls $u \sqsubseteq y$ für alle $u \in U$, so gilt $x \sqsubseteq y$
- $x \in X$ ist *kleinstes Element*, wenn $x \sqsubseteq y$ für alle $y \in X$.

Sätze

- Sind x, x' beides kleinste Elemente, so folgt $x = x'$. Man bezeichnet das kleinste element, wenn es existiert, mit dem Symbol $\bot$.
- Sind x, x' beides kleinste obere Schranken von $U \subseteq X$, so folgt $x = x'$. Man notiert das Supremum von U, wenn es existiert, mit $\sup U$ oder $\bigsqcup_{x \in U} x$.

Vollständigkeit

Eine partielle Ordnung X heißt *vollständig* (bzgl. ω-Ketten), wenn für jede ω-Kette $(x_i)_i$ das Supremum von $\{x_i \mid i \in \mathbb{N}$ existiert.

Man schreibt $\bigsqcup_i x_i$ für dieses Supremum.

Eine solche Ordnung heißt auf E. ω-complete partial order, kurz *ωcpo* oder noch kürzer *cpo*.

Bemerkung: Man kann den Begriff der Vollständigkeit auf *gerichtete Mengen* verallgemeinern, siehe [Kröger]. Wir brauchen das hier nicht.

Beispiele

Alle auf Folie 337 gegebenen Beispiele sind cpos.

Seien X, Y cpos. Die folgenden Konstruktionen liefern cpos.

- $Y_\perp$ definiert durch $Y_\perp = Y \cup \{\perp\}$ und $y \sqsubseteq y'$, falls $y = \perp$ oder $y \sqsubseteq y'$ in Y
- $[X \to Y] = \{f : X \to X \mid f \text{ stetig}\}$ mit $f \sqsubseteq g$, wenn $f(x) \sqsubseteq g(x)$ für alle $x \in X$.
-
- $X \times Y$ mit $(x, y) \sqsubseteq (x', y')$, wenn $x \sqsubseteq x'$ und $y \sqsubseteq y'$.
- X^* mit $[x_1; \ldots; x_m] \sqsubseteq [y_1; \ldots y_n]$, wenn $m = n$ und $x_i \sqsubseteq y_i$ für $i = 1, \ldots, m$.

Stetige Funktionen

Die Projektionen $X \times Y \to X$, $X \times Y \to Y$ sind stetig.

Sind $f : Z \to X$ und $g : Z \to Y$ stetig, so auch $h(z) = (f(z), g(z))$.

Die Applikation $(X \to Y) \times X \longrightarrow$ ist stetig.

Ist $f : X \times Y \to Z$ stetig, so ist $h : X \longrightarrow Y \to Z$ gegeben durch $h(x) = \mathbf{function}(y) f(x, y)$ wohldefiniert (liefert also stetige Funktionen) und außerdem selbst stetig.

Fixpunkte

Eine monotone Funktion $f : X \to Y$ heißt *stetig* (bzgl. ω-Ketten), wenn für jede Kette x_i gilt:

$$f(\bigsqcup_i x_i) = \bigsqcup_i f(x_i)$$

Man beachte, dass $f(x_i)$ wegen der Monotonie eine Kette bildet.

Fixpunkte

Sei $f : X \to X$ stetige Funktion. Ein Element $x \in X$ heißt *kleinster Fixpunkt* von f, wenn $f(x) = x$ und wenn immer $f(y) = y$, so folgt $x \leq y$.

Bemerkung: der kleinste Fixpunkt (falls er existiert) ist eindeutig bestimmt.

Fixpunktsatz von Kleene: Hat X ein kleinstes Element $\bot$, so hat jede stetige Funktion $f : X \to X$ einen kleinsten Fixpunkt, nämlich $x := \bigsqcup_i f^i(\bot)$.

Für diesen gilt zusätzlich folgendes: gilt $f(y) \sqsubseteq y$ für ein $y \in X$, so folgt $x \sqsubseteq y$.

Man schreibt $\mathrm{fix}(f)$ für den kleinsten Fixpunkt von f.

Satz: Die Funktion $\mathrm{fix} : (X \to X) \longrightarrow X$ ist selbst stetig.

Denotationelle Semantik mit cpos

Wir interpretieren nur wohltypisierte OCAML-Phrasen.

Zur Vereinfachung betrachten wir nur monomorphe Typen (keine Typvariablen).

Jedem Typen τ ordnen wir eine cpo $W(\tau)$ zu gemäß folgender Setzung:

$$W(\texttt{int}) = \{\text{Integer Konstanten}\} \text{ mit diskreter Ordnung}$$

$$W(\text{andere Basistypen}) = \text{analog}$$

$$W(\tau_1 \ \texttt{->} \ \tau_2) = W(\tau_1) \to W(\tau_2)_\perp$$

$$W(\tau_1 \ \texttt{*} \ \tau_2) = W(\tau_1) \times W(\tau_2)$$

$$W(\tau \ \texttt{list}) = W(\tau)^*$$

Andere Typformer werden ähnlich behandelt.

CPO der Umgebungen

Sei Γ eine Typzuweisung (endliche Funktion von Bezeichnern auf Typen).

Eine Umgebung U für Γ weist jedem Bezeichner in $\mathrm{dom}(\Gamma)$ ein Element $U(x) \in W(\Gamma(x))$ zu.

Die Umgebungen für Γ bilden eine cpo $\mathrm{Env}(\Gamma)$ mit $U \sqsubseteq U'$, wenn $U(x) \sqsubseteq U(x)$ für alle $x \in \mathrm{dom}(\Gamma)$.

Semantik von OCAML Phrasen

Zu jeder Programmphrase $\Gamma \triangleright t : \tau$ definieren wir eine stetige Funktion

$$W(t) : \mathrm{Env}(\Gamma) \to W(\tau)_\perp$$

durch Rekursion über den Aufbau von e wie folgt.

Wir schreiben wie üblich $W^U(t)$ statt $W(t)(U)$.

Auswerteregeln

- Ist c eine Konstante, so ist $W^U(c) = c$,
- Ist x ein Bezeichner, so ist $W^U(x) = w$, falls $<x, w> \in U$. Beachte: es muss x in $\mathrm{dom}(U)$ sein.
- Ist *op* ein Infixoperator aber nicht `||`, `&&`, welcher eine Basisfunktion $\oplus$ bezeichnet.

  Sind $W^U(t_1)$ und $W^U(t_2)$ beide ungleich $\perp$, so ist $W^U(t_1 \; op \; t_2) = W^U(t_1) \oplus W^U(t_2)$. Ansonsten ist das Ergebnis $\perp$.

  Einstellige Basisfunktionen wie `not` und `-` sind analog.

Auswertung von Funktionstermen

- Sei t eine Funktionsanwendung der Form $t_1\ t_2$. Es sei $W^U(t_1)$ die Funktion f und $W^U(t_2) = w$. Ist auch nur eines der beiden gleich $\bot$, so ist $W^U(t) = \bot$.

  Ansonsten ist $W^U(t) = f(w)$. Dies kann trotzdem noch $= \bot$ sein, da ja $W(\tau_1\texttt{->}\tau_2) = W(\tau_1 \to W(\tau_2)_\bot$.

- Sei $t = \texttt{function}\ \ x\ \ \texttt{->}\ \ t'$ und $\Gamma \rhd t : \tau_1\texttt{->}\tau_2$.

  Es ist $W^U(t)$ diejenige Funktion, die $w \in W(\tau_1)$ auf $W^{U+\{<x,w>\}}(t') \in W(\tau_2)_\bot$ abbildet.

- Die alternativen Notationen für Funktionsdefinitionen (`fun`, `let`) haben analoge Bedeutung.

Auswertung von `if` und `let`

- Sei t ein bedingter Term der Gestalt `if` t_1 `then` t_2 `else` t_3. Ist $W^U(t_1) = \mathit{true}$, so ist $W^U(t) = W^U(t_2)$. Beachte: $W^U(t_3)$ kann dann $= \bot$ sein.

  Ist $W^U(t_1) = \mathit{false}$, so ist $W^U(t) = W^U(t_3)$. Beachte: $W^U(t_2)$ kann dann $= \bot$ sein.

  Ist dagegen $W^U(t_1) = \bot$, so auch $W^U(t)$.

- Sei t von der Form `let` x `=` t_1 `in` t_2.

  Sei $W^U(t_1) = w \neq \bot$. Dann ist $W^U(t) = W^{U+\{<x,w>\}}(t_2)$.
  Ansonsten ist $W^U(t) = \bot$.

Tupel und Boole'sche Operatoren

- Sei $t = (t_1, \ t_2)$. Seien weiter
  $W^U(t_1) = w_1 \neq \perp W^U(t_2) = w_2 \neq \perp$. Dann ist $W^U(t) = (w_1, w_2)$.
  Ansonsten ist $W^U(t) = \perp$.
- $W^U(t_1 \ \texttt{||} \ t_2) = W^U(\texttt{if} \ t_1 \ \texttt{then true else} \ t_2)$
- $W^U(t_1 \ \texttt{\&\&} \ t_2) = W^U(\texttt{if} \ t_1 \ \texttt{then} \ t_2 \ \texttt{else false})$.

Semantik von `let rec`

Eine rekursive `let`-Bindung

$$\texttt{let rec } f\ x \texttt{ = } t$$

bindet f an die Funktion

$$\mathrm{fix}(\mathbf{function}F.\mathbf{function}x.W^{U+\{<x,w>,<f,F>\}}(t))$$

Das gilt analog für Phrasen mit `let rec ...in` und für Funktionen mit mehreren Argumenten.

Man beachte, dass $W(\tau_1 \texttt{->} \tau_2)$ ein kleinstes Element besitzt, nämlich $\mathbf{function}\ w.\bot$. Somit existiert dieser kleinste Fixpunkt.

Beispiel: Fakultät

```
let rec fakt n = if n = 0 then 1 else n * fakt(n-1)
```

Sei $F = W^{\emptyset}(\texttt{fakt})$.

Es ist $F = \mathrm{fix}(\Phi)$, wobei

$$\Phi(f) = \textbf{function}\ w.W^{U+\{<\texttt{n},w>,<\texttt{fakt},f>\}}(\texttt{if n = 0 then 1 else n * fakt(n-1)})$$

Es gilt

$$\Phi(f) = \textbf{function}w.\textbf{if}\ w = 0\ \textbf{then}\ 1\ \textbf{else}\ w \cdot f(w-1)$$

Bestimmung des Fixpunkts

Wir berechnen $\Phi^i(\bot)$, wobei $\bot \in \mathbb{Z} \to \mathbb{Z}_\bot$ die Funktion "konstant $\bot$" ist.

Es ergibt sich folgende Wertetabelle:

	< 0	0	1	2	3	4	5	> 5
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	
$\Phi(\bot)$	$\bot$	1	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$\Phi^2(\bot)$	$\bot$	1	1	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$\Phi^3(\bot)$	$\bot$	1	1	2	$\bot$	$\bot$	$\bot$	$\bot$
$\Phi^4(\bot)$	$\bot$	1	1	2	6	$\bot$	$\bot$	$\bot$
$\Phi^5(\bot)$	$\bot$	1	1	2	6	24	$\bot$	$\bot$

Wir sehen, dass $\bigsqcup_i \Phi^i(\bot) = \mathbf{function} w.w!$.

Operationelle Adäquatheit

Es gilt der folgende Satz:

Haben zwei Terme die gleiche denotationelle Semantik, so kann man in einem beliebigen Programm den einen Term durch den anderen ersetzen, ohne das Verhalten des Programms im Sinne der operationellen Semantik zu verändern.

Insofern stellt die operationelle Semantik eine korrekte Implementierung der denotationellen Semantik dar.

Will man umgekehrt die operationelle Semantik als “maßgeblich” auffassen, so bedeutet der Satz, dass die denotationelle Semantik ein korrektes Schlussprinzip für Austauschbarkeit von Programmteilen liefert.