5 KFP: Invarianz der Terminierung, Standardisierungssatz

Wir behandeln jetzt die Sprache KFP etwas formaler. Das Ziel dieses Kapitels ist der Nachweis, dass Terminierung von Ausdrücken erhalten bleibt, wenn man reduziert. Genauer, dass im Falle $t \to t'$ die Terminierung sich nicht verändert; d.h. $t\Downarrow \Leftrightarrow t'\Downarrow$. Diese erfordert die Betrachtung beider Implikationen $t\Downarrow \Rightarrow t'\Downarrow$ und $t\Downarrow \Leftarrow t'\Downarrow$, die verschiedene Methoden verwenden.
Eine verwandte, aber doch leicht andere Aussage, ist der Nachweis, dass jeder Ausdruck, der sich mit irgendeiner Folge von Reduktion mit den Kalkülregeln auf eine WHNF reduzieren läßt, auch eine terminierende no-Reduktion hat.
Das ganze ist technisch etwas aufwendig, aber doch elementar. Da diese Aussagen die Basis für die Korrektheit von Programmtransformationen sind, lohnt es sich auch, diese etwas genauer durchzugehen.
Wir benötigen dafür die sogenannte 1-Reduktion (siehe [Bar84]), deren Bedeutung die parallele Reduktion ist. D.h. zwei Terme s, t stehen in der 1-Relation, wenn s parallel auf t reduzierbar ist. Dieser Parallelitätsbegriff muß formal definiert werden, da es nicht ganz offensichtlich ist, wie er gemeint ist. Betrachte z.B.

Beispiel 5.1. $(\lambda x.x\ x)((\lambda y\ .\ y)a)$. Es gibt zwei Redexe die man unabhängig voneinander reduzieren kann. Z.B. ist dieser Term parallel reduzierbar zu: $a\ a$.
Aber: der Term $(\lambda x.x\ x)(\lambda y\ .\ y)$ ist nur 1-reduzierbar zu $(\lambda y\ .\ y)\ (\lambda y'\ .\ y')$, und nicht zu $(\lambda y'\ .\ y')$. Diese Reduktionen sind sequentiell, denn der zweite Redex entsteht erst durch die Reduktion.

Definition 5.2. *Sei $\to_1$ die folgende Relation auf Ausdrücken:*

- $s \to_1 s$
- *Wenn* $s \to_1 s_1$ *und* $t \to_1 t_1$, *dann* $(s\ t) \to_1 (s_1\ t_1)$.
- *Wenn* $s \to_1 s_1$, *dann* $(\lambda\ x\ .\ s) \to_1 (\lambda\ x\ .\ s_1)$.
- *Wenn* $s_i \to_1 s_i'$ *und* $t \to_1 t'$, *dann*
  $\texttt{case}_T\ t\ ((c\ x_{1,1} \dots x_{1,n_1}) \to s_1) \dots ((c\ x_{m,1} \dots x_{m,n_m}) \to s_m)$
  $\to_1 \texttt{case}_T\ t'\ ((c\ x_{1,1} \dots x_{1,n_1}) \to s_1') \dots ((c\ x_{m,1} \dots x_{m,n_m}) \to s_m')$.
- *Wenn* $s \to_1 s_1$, $t \to_1 t_1$, *dann* $(((\lambda\ x\ .\ s))\ t) \to_1 s_1[t_1/x]$
- *Wenn* $s_i \to_1 s_i', t \to_1 t'$, *dann*
  $(\texttt{case}_T(c\ s_1\ \dots\ s_n) \dots ((c\ x_1 \dots x_n) \to t) \dots)$
  $\to_1 t'[s_1'/x_1, \dots, s_n'/x_n]$, *wenn c ein Konstruktor des Typs T ist und $ar(c) = n$.*

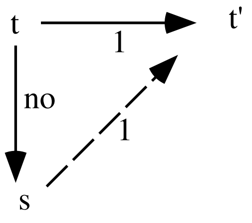
Die 1-Relation kann aufgelöst werden in eine Folge von normalen Reduktionen:

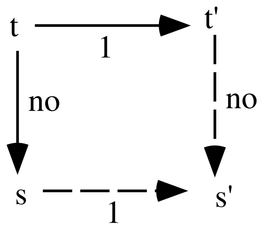
Lemma 5.3. *Wenn $s \to_1 t$ gilt, dann auch $s \xrightarrow{*} t$.*

Beweis. Das kann man mit Induktion nach der Struktur des Terms s zeigen.
Als Beispiel zeigen wir wie das im Fall der Beta-reduktion geht: Angenommen, $s = ((\lambda x.s_1)\ s_2)$ und $s_1 \to_1 t_1$, $s_2 \to_1 t_2$ und $t = t_1[t_2/x]$. Dann gilt mit

Induktion, dass $s_1 \xrightarrow{*} t_1$, $s_2 \xrightarrow{*} t_2$, also auch $((\lambda x.s_1)\ s_2) \xrightarrow{*} ((\lambda x.s_1)\ t_2) \xrightarrow{*} ((\lambda x.t_1)\ t_2) \to t_1[t_2/x]$, also insgesamt $s \xrightarrow{*} t$. □

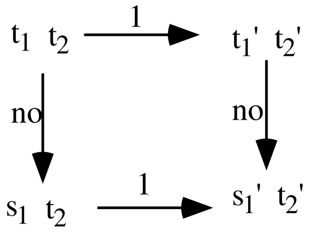
Lemma 5.4. *Sei t ein (evtl. offener) Ausdruck. Wenn $t \xrightarrow{no} s$ und $t \to_1 t'$, dann ist entweder $s \to_1 t'$, oder es existiert ein Term s', so dass $s \to_1 s'$, und $t' \xrightarrow{no} s'$.*





Beweis. Dies zeigt man mit Induktion über die Struktur des Ausdrucks: Dazu braucht man eine Fallunterscheidung:

– Sei $t \equiv t_1\ t_2$ und t_1 ist keine Abstraktion. Dann ändert die Normalordnung den Ausdruck t_2 nicht. Für den Ausdruck t_1 ist die Tiefe des Normalordnungsredex kleiner, also gilt mit $t_1 \xrightarrow{no} s_1$, auch: $t_1\ t_2 \xrightarrow{no} s_1\ t_2$. Für $t_1\ t_2 \to_1 t_1'\ t_2'$ gilt $t_1 \to_1 t_1'$ und $t_2 \to_1 t_2'$. Mit Induktion gibt es jetzt zwei Fälle:
  - $s_1 \to_1 t_1'$. Dann gilt auch $s_1\ t_2 \to_1 t_1' t_2'$.
  - Es gibt ein s_1' mit $s_1 \to_1 s_1'$ und $t_1' \xrightarrow{no} s_1'$.
    Zusammengesetzt ergibt das: $s_1\ t_2 \to_1 s_1'\ t_2'$ und $t_1'\ t_2' \xrightarrow{no} s_1'\ t_2'$.



– $t \equiv t_1\ t_2$ und t_1 ist eine Abstraktion $\lambda x\ .\ t_3$. Dann ist die Normalordnungsreduktion gerade $t_1\ t_2 \xrightarrow{no} t_3[t_2/x]$, wobei wir die Konvention über disjunkte (gebundene) Variablennamen annehmen. Es gibt zwei Fälle für die 1-Relation:
  Wenn $t_1\ t_2 \to_1 t_1'\ t_2'$, dann können wir auch schreiben $(\lambda x\ .\ t_3)\ t_2 \to_1 (\lambda x\ .\ t_3')\ t_2'$ mit $t_3 \to_1 t_3', t_2 \to_1 t_2'$. Damit gilt $t_3[t_2/x] \to_1 t_3'[t_2'/x]$ und $(\lambda x\ .\ t_3')\ t_2' \xrightarrow{no} t_3'[t_2'/x]$.
  Wenn $(\lambda x\ .\ t_3)\ t_2 \to_1 t_3'\ [t_2'/x]$, dann gilt $t_3\ [t_2/x] \to_1 t_3'\ [t_2'/x]$, und wir haben die Behauptung gezeigt.
– Wenn $t \equiv \texttt{case}\ t_1\ alt_1 \ldots alt_n$, und dieser Ausdruck ist kein Normalordnungsredex, dann kann man Induktion über die Struktur wie im ersten Fall machen.

- Wenn $t \equiv$ `case` $t_1\ alt_1 \dots alt_n$ und der Ausdruck ist ein Normalordnungsredex, dann ist die Normalordnungsreduktion `case` $(c\ t_1 \dots t_n)\ \dots ((c\ x_1 \dots x_n) \to t_{n+1}) \dots) \xrightarrow{no} t_{n+1}[t_1/x_1, \dots t_n/x_n]$.
  Die 1-Reduktion kann sein: `case` $(c\ t_1 \dots t_n)\ \dots ((c\ x_1 \dots x_n) \to t_{n+1}) \dots) \to_1$ `case` $(c\ t_1' \dots t_n')\ \dots ((c\ x_1 \dots x_n) \to t_{n+1}')$. In diesem Fall ist die Vertauschbarkeit analog zu den anderen Fällen.
  Wenn die 1-Reduktion so aussieht: `case` $(c\ t_1 \dots t_n)\ \dots ((c\ x_1 \dots x_n) \to t_{n+1}) \dots) \to_1 t_{n+1}'[t_1'/x_1, \dots, t_n'/x_n]$, dann ist auch: $t_{n+1}[t_1/x_1, \dots, t_n/x_n], \to_1 t_{n+1}'[t_1'/x_1, \dots, t_n'/x_n]$.

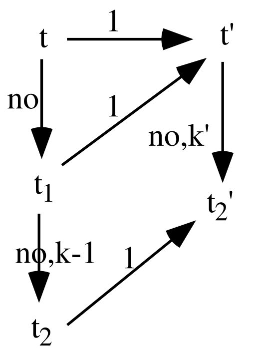
□

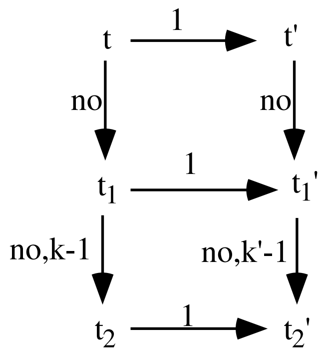
Aussage 5.5. *Sei t ein (evtl offener) Ausdruck. Wenn $t \xrightarrow{no,k} t_2$ und t_2 ist eine WHNF, und $t \to_1 t'$. Dann terminiert die no-Reduktion von t' und es gilt $t' \xrightarrow{no,k'} t_2'$, $t_2 \to_1 t_2'$, wobei $k \geq k'$ und t_2' eine WHNF ist.*

Beweis. Induktion nach k, der Anzahl der Normalordnungsreduktionen. Im Basisfall muß man nur noch beachten, dass aus $t \to_1 t'$ und t ist WHNF folgt, dass t' ebenfalls eine WHNF ist.

Im Falle $k > 0$ zerlegen wir die no-Reduktion in $t \xrightarrow{no} t_1 \xrightarrow{no,k-1} t_2$. Nach Lemma 5.4 gibt es zwei Fälle:

- $t_1 \to_1 t'$. Induktion zeigt die Existenz einer WHNF t_2' mit $t_2 \to_1 t_2'$ und $t' \xrightarrow{no,k'} t_2'$ und $k - 1 \geq k'$. Dann gilt auch $k \geq k'$ und die Behauptung des Lemmas ist gezeigt.
- Es gibt ein t_1' mit $t' \xrightarrow{no} t_1'$ und $t_1 \to_1 t_1'$. Induktion nach k zeigt dann die Existenz einer WHNF t_2' mit $t_1' \xrightarrow{no,k'-1} t_2'$ und $t_2 \to_1 t_2'$ und $k - 1 \geq k' - 1$. Zusammensetzen der Diagramme ergibt $t' \xrightarrow{no,k'} t_2'$ und $t_2 \to_1 t_2'$ und $k \geq k'$.





□

Da jede 1-Schritt-Reduktion auch als 1-Relation darstellbar ist, folgt unmittelbar:

Satz 5.6. *Sei t ein (evtl. offener) Ausdruck. Wenn $t\Downarrow$, d.h. $t \xrightarrow{no,*} t_1$ und t_1 ist eine WHNF, und $t \to t'$ mit einer anderen Reduktion. Dann gilt $t'\Downarrow$.*

Jetzt fehlt noch die Richtung: $(t \to t' \wedge t'\Downarrow) \Rightarrow t\Downarrow$. Die Idee hier ist, eine Reduktion zur WHNF, die aus Normalordnungsreduktionen und 1-Reduktionen besteht, in eine Normalordnungsreduktion zu verwandeln.

Definition 5.7. *Sei t ein (evtl. offener) Term. Die Reduktion $t \to_1 s$ ist* intern*, d.h. ohne anteilige Normalordnungsreduktion, wenn sie nicht von einer der Formen ist:*

- $R[((\lambda x\ .\ t_1)\ t_2)] \to_1 R'[t_1'[t_2'/x]]$ *wobei R ein Reduktionskontext ist, $t_1 \to_1 t_1', t_2 \to_1 t_2', R \to_1 R'$, wobei man $\to_1$ für Kontexte wie für einen Term mit einer extra Konstanten definiert.*
- $t = R[(\mathtt{case}_T\ (c\ t_1 \ldots t_n)\ \ldots((c\ x_1 \ldots x_n) \to t_{n+1}))] \to_1 R'[t'_{n+1}[t'_1/x_1,\ldots,t'_n/x_n]]$ *wobei R ein Reduktionskontext ist und $t_i \to_1 t_i', R \to_1 R'$.*

□

Man kann die Anzahl der notwendigen Einzelreduktionen in der parallelen Reduktion $r_1 \to_1 r_2$ durch Markieren der Unterterme des Terms r_1 mit dieser Anzahl festhalten. Diese Anzahl entspricht der Anzahl Reduktionen, wenn man diese "von oben" her abarbeitet. Diese Anzahl kann man berechnen anhand der syntaktischen Form von r_1 und der Änderungen, die die 1-Reduktion bewirkt. Sei φ diese Markierung, geschrieben als Abbildung von Untertermen auf die Anzahl. Eigentlich ist das eine Abbildung der ganzen Reduktion.

- Wenn $(s\ t) \to_1 s'\ t'$, dann markiere $(s\ t)$ mit $\varphi(s) + \varphi(t)$
- Wenn $(\lambda x\ .\ s) \to_1 (\lambda x\ .\ s')$, dann markiere $(\lambda x\ .\ s)$ mit $\varphi(s)$.
- Wenn $(\mathtt{case}\ s\ (p_1 \to t_1)\ldots(p_n \to t_n)) \to_1 (\mathtt{case}\ s'\ (p_1 \to t_1')\ldots(p_n \to t_n'))$, dann markiere mit $\varphi(s) + \Sigma\ \varphi(t_i)$.
- Wenn $(\lambda x\ .\ s)\ t \to_1 s'[t'/x]$, dann markiere mit $1 + \varphi(s) + \varphi(t) * (\#(x,s'))$ wobei $\#(x,s)$ die Anzahl der freien Vorkommen von x in s ist.
- Wenn $(\mathtt{case}\ (c\ s_1 \ldots s_n)\ \ldots((c\ x_1 \ldots x_n) \to t)\ldots) \to_1 t[s_1/x_1,\ldots,s_n/x_n]$, dann markiere mit $1 + \varphi(t) + (\Sigma\ (\varphi(s_i') * (\#(x_i,t))))$.

Lemma 5.8. *Sei t ein (evtl. offener) Ausdruck. Wenn $t \to_1 s$, dann kann man die Reduktion zerlegen in: $t \xrightarrow{no,*} u \to_1 s$, wobei die Reduktion $u \to_1 s$ intern ist.*

Beweis. Dass man eine nicht-interne Reduktion $t \to_1 s$ in $t \xrightarrow{no} u_1 \to_1 s$ zerlegen kann, folgt aus der Definition der Reduktion und aus der Definition von intern. Dies kann man iterieren, bis die Reduktion $u_1 \to_1 s$ intern ist. Allerdings ist nicht ganz offensichtlich, ob es terminiert. Dazu muß man die Anstrengung etwas erhöhen: Die Idee ist, dass die Anzahl φ der notwendigen Einzelreduktionen vermindert wird: Es gibt zwei Fälle (beta und case):

- Wenn die erste Zerlegung so aussieht: $R[(\lambda x\ .\ s)\ t] \xrightarrow{no} R[s[t/x]] \to_1 R'[s'[t'/x]]$, dann ist die Zahl der Reduktionen (φ) in der Reduktion $R[s[t/x]] \to_1 R'[s'[t'/x]]$ um 1 vermindert.
- Wenn die Zerlegung so aussieht: $R[(\mathtt{case}\ (c\ s_1 \ldots s_n)\ \ldots ((c\ x_1 \ldots x_n) \to t) \ldots)] \xrightarrow{no} R[t[s_1/x_1, \ldots, s_n/x_n]] \to_1 R'[t'[s_1'/x_1, \ldots, s_n'/x_n]]$, dann folgt ebenfalls aus der obigen Definition, dass die Anzahl der Reduktionen mindestens um 1 vermindert ist. Die Verminderung kann größer sein, denn eine case-Reduktion kann ganze Unterterme löschen.

Insgesamt heißt das, dass das Abspalten von no-Reduktionen terminiert. □

Lemma 5.9. *Sei t ein (evtl. offener) Ausdruck. Wenn $t \to_1 u \xrightarrow{no} s$, dann kann man die Reduktionen vertauschen: $\exists u' : t \xrightarrow{no,+} u' \to_1 s$, so dass die Reduktion $u' \to_1 s$ intern ist.*

Beweis. Die Reduktionsfolge $t \to_1 u \xrightarrow{no} s$ kann man zerlegen in: $t \xrightarrow{no,*} t' \to_1 u \xrightarrow{no} s$, so dass $t' \to_1 u$ intern ist. Beachte, dass die 1-Relation auch trivial sein kann. Es gibt nur die folgenden zwei Fälle für die Reduktion $t' \to_1 u \xrightarrow{no} s$:

1. Der erste Fall ist, dass $R[(\lambda x\ .\ t_1)\ t_2] \to_1 R'[(\lambda x\ .\ t_1')\ t_2'] \xrightarrow{no} R'[t_1'[t_2'/x]]$, wobei R Reduktionskontext, $t_1 \to_1 t_1', t_2 \to_1 t_2', R \to_1 R'$, wobei man $\to_1$ für Kontexte wie für einen Term mit einer extra Konstanten definiert (eigentlich braucht man dafür Induktion nach der Termstruktur). In diesem Fall gilt: $R[(\lambda x\ .\ t_1)\ t_2] \xrightarrow{no} R[t_1[t_2/x]] \to_1 R'[t_1'[t_2'/x]]$.
2. Der zweite Fall ist $t = R[(\mathtt{case}_T\ (c\ t_1 \ldots t_n)\ \ldots ((c\ x_1 \ldots x_n) \to t_{n+1}))] \to_1 R'[(\mathtt{case}_T\ (c\ t_1' \ldots t_n')\ \ldots ((c\ x_1 \ldots x_n) \to t_{n+1}'))] \xrightarrow{no} R'[t_{n+1}'[t_1'/x_1, \ldots, t_n'/x_n]]$ wobei R Reduktionskontext, $t_i \to_1 t_i', R \to_1 R'$. In diesem Fall gilt wie im ersten $t \xrightarrow{no} R[t_{n+1}[t_1/x_1, \ldots, t_n/x_n]] \to_1 R'[t_{n+1}'[t_1'/x_1, \ldots, t_n'/x_n]]$.

D.h. in diesem Fall ist die Reduktion vertauschbar, und wir erhalten: $t \xrightarrow{no,*} t_1 \xrightarrow{no} t_2 \to_1 s$. Wenn die letzte Reduktion nicht intern ist, dann kann man mit Lemma 5.8 den Normalordnungsanteil abspalten. □

Lemma 5.10. *Sei s ein (evtl. offener) Ausdruck. Eine Folge $s \to_1 s_2 \xrightarrow{no} \ldots \xrightarrow{no} s_n$ wobei s_n eine WHNF ist, kann man umändern in eine Reduktion, die nur Normalordnungs-Reduktionen verwendet, um zu einer WHNF zu kommen. Genauer: in eine Reduktion $s \xrightarrow{no,*} s_{n-1}' \to_1 s_n$, bei der s_{n-1}' eine WHNF ist.*

Beweis. Das Lemma 5.9 kann man zum Verschieben der 1-Reduktion nach rechts verwenden. $s \to_1 s_2 \xrightarrow{no} \ldots \xrightarrow{no} s_n$ wird damit im ersten Schritt zu $s \xrightarrow{no,*} s_2' \to_1 s_3 \ldots \xrightarrow{no} s_n$. Schiebt man die 1-Reduktion weiter nach rechts, so erhält man schließlich: $s \xrightarrow{no,*} s'_2 \xrightarrow{no,*} s_3 \dots s'_{n-1} \rightarrow_1 s_n$. Die letzte Reduktion ist intern, also ist s'_{n-1} eine WHNF.

□

Satz 5.11. *(Standardisierung). Sei t ein (evtl. offener) Ausdruck. Wenn* $t \xrightarrow{*} t_1$ *mit beliebigen (Beta) und (case)-Reduktionen, wobei* t_1 *eine WHNF ist, dann existiert eine WHNF* t_{NF}*, so dass* $t \xrightarrow{no,*} t_{NF}$*, und* $t_{NF} \xrightarrow{*} t_1$.

Beweis. Jede Reduktion ist auch eine Folge von 1-Reduktionen. Danach kann man Lemma 5.10 mehrfach anwenden. □

Satz 5.12. *(Invarianz der Terminierung) Seien* t, t' *(evtl. offene) Ausdrücke mit* $t \rightarrow t'$*. Dann gilt* $t\Downarrow \Leftrightarrow t'\Downarrow$

Beweis. Das folgt aus Satz 5.11 (Standardisierung) und Satz 5.6. □

5.1 Exkurs: verallgemeinerte Induktion

Eine partielle Ordnung $<$ auf einer Menge M ist *fundiert (well-founded)*, gdw. jede absteigende Kette endlich ist.
Wenn man eine fundierte Ordnung $<$ auf einer Menge M mit Menge der kleinsten Elemente M_{min} hat, dann kann man folgendes Induktionsschema verwenden:

$$\begin{array}{lr} ((\forall m \in M_{min} : P(m)) & \text{Basis} \\ \wedge\ (\forall x : (\forall y. y < x \Rightarrow P(y)) \Rightarrow P(x))) & \text{Induktionsschluß} \\ \hline \Rightarrow \forall x. P(x) & \end{array}$$

Eine lineare fundierte Ordnung, die wir verwenden wollen, ist die lexikographische Ordnung auf Tupeln. Es gilt:

> Wenn die Ordnung $<$ auf M fundiert ist, dann auch die lexikographische Ordnung auf den Tupeln M^n, wobei $(a_1, \dots, a_n) < (b_1, \dots, b_n)$ gdw. für einen Index $i : a_1 = b_1, \dots, a_{i-1} = b_{i-1}, a_i < b_i$

Die Ordnung $<$ auf den natürlichen Zahlen ist fundiert. Also auch die lexikographische Ordnung auf Tupeln über natürlichen Zahlen.

6 Kontextuelle Gleichheit von Ausdrücken in KFP

Gleichheit von Ausdrücken s, t kann man annehmen, wenn s und t sich gleich verhalten (Verhaltensgleichheit). Da die Erkennung des Verhaltens ein formales Beobachten erfordert, benutzen wir das Kriterium der Terminierung bzw. Nichtterminierung als Ergebnis der theoretischen Beobachtung. Natürlich kann

man Nichtterminierung nicht praktisch bzw. effektiv beobachten, insofern ist das Beobachtungskriterium nicht effektiv. Die formale Definition hilft aber beim Erkennen und Nachweisen der Korrektheit von Programmtransformationen, wie wir sehen werden.
Man sieht (offene und geschlossene) Ausdrücke s, t als gleich an, wenn Programme ihr Terminierungsverhalten niemals ändern, wenn man s durch t ersetzt. Dieses Kriterium erscheint zunächst schwach, aber da man alle Programme und alle Stellen innerhalb der Programme ausprobieren kann, ergibt sich ein starkes Kriterium.
Wenn man Beobachten im engeren Sinne auslegt, könnte man auch die Effizienz eines Programmes als Kriterium nehmen (Anzahl der Schritte). Das wäre eine stärkere Forderung, die wir aber hier nicht betrachten, da sich dann fast nichts mehr nachweisen läßt.
Man definiert zunächst eine Approximationsordnung $s \leq_c t$, die bedeutet, dass t besser terminiert als s, oder, dass s weniger Information liefert als t.

Definition 6.1. *(Kontextuelle Approximation) Seien s, t KFP-Ausdrücke und $C[.]$ ein Kontext.*

$$s \leq_c t$$
$$gdw.$$
$$\forall C[] : C[s]\Downarrow \Rightarrow C[t]\Downarrow$$

$$s \sim_c t \text{ } gdw \text{ } s \leq_c t \wedge t \leq_c s$$

Die Ordnung $\leq_c$ nennen wir kontextuelle Approximation*, und $\sim_c$* kontextuelle Gleichheit*.*

Ein großer Vorteil dieser Definition ist, dass man sich ganz auf die definierte Sprache und die Auswertung zurückziehen kann, ohne die Hilfe einer externen Semantik in Anspruch zu nehmen.
Ein kleiner Nachteil ist, dass diese Definition der Gleichheit von Ausdrücken leider nicht ganz "kanonisch" ist. Eine Diskussion dazu ist in einem späteren Kapitel. Auch ist es möglich, dass die Gleichheit sich ändert, wenn man die Sprache ändert bzw. erweitert.

Beispiel 6.2. Mit $\sim_c$ kann man z.B. $(S\ 0)$ von `True` unterscheiden, indem man den Kontext `case_Bool .  (True -> True) (False -> True)` verwendet. `case_Bool (S 0)  (True -> True) (False -> True)` hat keine WHNF, aber `case_Bool True  (True -> True) (False -> True)` ist reduzierbar zu `True`.

Eine wichtige und (im Gegensatz zu anderen Definitionen) leicht zu zeigende Eigenschaft ist, dass sich durch Ersetzung eines Unterausdrucks e durch einen kontextuell gleichen Ausdruck e' ein Programm in ein kontextuell äquivalentes verwandelt.

Aussage 6.3. *$\leq_c$ und $\sim_c$ sind stabil gegen Einsetzung in Kontexte und $\leq_c$ ist eine Präordnung, während $\sim_c$ eine Äquivalenzrelation ist. Zusammengefaßt: $\leq_c$ ist eine Präkongruenz und $\sim_c$ ist eine Kongruenzrelation auf Ausdrücken.*

Beweis. Offensichtlich ist $\leq_c$ transitiv und $\sim_c$ ist eine Äquivalenzrelation. Wir zeigen, dass $\leq_c$ stabil gegen Einsetzung in Kontexte ist. Sei dazu $s \leq_c t$. Wir zeigen dass für einen beliebigen Kontext C: $C[s] \leq_c C[t]$ gilt. Sei D ein Kontext und sei $D' := D[C[]]$. Aus $s \leq_c t$ folgt dann dass $D[C[s]]\Downarrow \Rightarrow D[C[t]]\Downarrow$. Da das für alle Kontexte D gilt, folgt die Behauptung. $\square$

Lemma 6.4. *Für alle Variablen x gilt: $x \sim_c x$, aber für verschiedene Variablen x, y gilt $x \not\sim_c y$*

Beweis. Betrachte den Kontext $C = (\lambda y\ .\ (\lambda x\ .\ []))\ bot$ `Nil`. Dann reduziert $C[x]$ zu `Nil`, während $C[y]$ nicht terminiert. $\square$

Der nächste Satz ist der erste Satz über Korrektheit von Programmtransformationen. Er sagt aus, dass man die Reduktionsregeln Beta und Case an beliebiger Stelle (vorwärts und rückwärts) in einem Programm verwenden kann, um das Programm (zur Compilezeit) zur verändern, bzw. zu optimieren, ohne dass man einen Fehler macht. Dies nennt man auch *partielle Auswertung*.

Satz 6.5. *Sei $s \to t$ für zwei (auch offene) KFP-Ausdrücke. Dann gilt $s \sim_c t$.*

Beweis. Sei C ein Kontext. Dann gilt $C[s] \to C[t]$, da Reduktionen überall erlaubt sind. Nach Satz 5.12 ist $C[s]\Downarrow \Leftrightarrow C[t]\Downarrow$. Da das für alle Kontexte gilt, gilt auch $s \sim_c t$. $\square$

Man kann auch fordern, dass die Kontexte die Terme s, t schließen:

Aussage 6.6. *In KFP gilt: $s \leq_c t$ gdw. $\forall C$: wenn $C[s], C[t]$ geschlossen, dann $C[s]\Downarrow \Rightarrow C[t]\Downarrow$.*

Beweis. Die eine Richtung ist klar.
Es gelte $\forall C$: wenn $C[s], C[t]$ geschlossen, dann $C[s]\Downarrow \Rightarrow C[t]\Downarrow$. Sei C irgendein Kontext, so dass $C[s]\Downarrow$, aber nicht $C[t]\Downarrow$. Dann erweitern wir den Kontext C so dass alle freien Variablen in $C[s], C[t]$ eingefangen werden, indem wir für die freien Variablen $x_1, \ldots x_n$ den Kontext $D[] := (\lambda x_1, \ldots, x_n.C[])bot \ldots bot$ nehmen.
Dann gilt weiterhin $D[s]\Downarrow$, denn eine WHNF ist unbeeinflusst von den Einsetzungen in die freien Variablen. Es gilt auch noch $D[t]\Uparrow$, denn entweder ist die Normalordnungsreduktion nicht terminierend, oder ergibt einen Typfehler, was nicht durch die Einsetzung beeinflusst ist, oder bei der Reduktion von $C[t]$ gerät eine eine freie Variable in einen Reduktionskontext. Diese ist aber durch bot ersetzt, somit gilt $D[t]\Uparrow$, ein Widerspruch. $\square$

Wir können für Ausdrücke, die kontextuell gleich sind, mehr über die Struktur der zugehörigen WHNF sagen.

Definition 6.7. *Ein* Konstruktorterm *ist ein Ausdruck, der nur aus Konstruktoren besteht.*

Beachte, dass damit endliche Konstruktorterme gemeint sind, denn Ausdrücke sind immer endlich.

Aussage 6.8. *Sei* $s \sim_c t$, $s \xrightarrow{*} (c_1\ s_1 \dots s_n)$ *und* $t \xrightarrow{*} (c_2\ t_1 \dots t_m)$.
Dann gilt:

1. $ar(c_1) \geq n$ *gdw.* $ar(c_2) \geq m$ *und*

2. $ar(c_1) \geq n$ *impliziert dass* $c_1 = c_2, m = n$ *und für alle i:* $s_i \sim_c t_i$.

Beweis. Wenn $ar(c_1) \geq n$, dann hat s im leeren Kontext eine WHNF, also auch t, insbesondere gilt $ar(c_2) \geq m$. Dasselbe gilt umgekehrt.
Wenn $ar(c_1) \geq n$, dann ist auch $ar(c_2) \geq m$.
Es gilt $c_1 = c_2$ und $m = n$ sein:
Dies sieht man, wenn man die Ausdrücke in den richtigen Kontext einsetzt:

```
C :=  case_T []  ((c_1 ...) -> True) ((c_i ... ) -> bot)  ...
```

dessen Resultat für alle Konstruktoren $\neq c_1$ gleich *bot* ist. Dann ist $C[s]\Downarrow$, also auch $C[t]\Downarrow$, was nur sein kann wenn $c_1 = c_2$ und $m = n$.
Die Äquivalenz der Komponenten erhält man, wenn man einen Kontext verwendet, der die Komponenten projiziert:

$C_i =$ `case_T [] ... ((c_1  x_1 ... x_i ... x_k) -> x_i) ...`

Das ergibt mit Satz 6.5: $C_i[s] \sim_c s_i$. Da aus $s \sim_c t$ auch $C_i[s] \sim_c C_i[t]$ folgt und $C_i[t] \sim_c t_i$ ist, erhält man $s_i \sim_c t_i$ für alle i. □

Satz 6.9. *Erhält man für einen Ausdruck mittels zwei verschiedener Auswertungen jeweils Konstruktorterme, so sind diese gleich.*
Formal: $s \xrightarrow{*} s_1$, $s \xrightarrow{*} s_2$ *wobei* s_1, s_2 *Konstruktorterme sind, dann gilt* $s_1 = s_2$.

Beweis. Folgt aus der Aussage 6.8 mit Induktion über die Tiefe des Konstruktorausdrucks. □

Jetzt können wir auch noch zeigen, dass man für Ausdrücke mit freien Variablen eingeschränktere Kriterien für die kontextuelle Äquivalenz hat:

Lemma 6.10. *Seien* s, t *KFP-Ausdrücke mit den freien Variablen* $x_1, \dots, x_n$ *und sei* ρ *eine Umbenennung der Variablen* x_i. *D.h. eine Injektion mit* $\rho(x_)) = y_i$. *Dann gilt* $s \leq_c t$ *gdw.* $s\rho \leq_c t\rho$.

Aussage 6.11. *Seien* s, t *KFP-Ausdrücke mit den freien Variablen* $x_1, \dots, x_n$. *Dann gilt* $s \leq_c t$ *gdw.* $\lambda x_1, \dots, x_n.s \leq_c \lambda x_1, \dots, x_n.t$.

Beweis. Die eine Richtung ist klar.
Sei $\lambda x_1, \ldots, x_n s \leq_c \lambda x_1, \ldots, x_n t$, und sei C ein Kontext, so dass $C[s], C[t]$ geschlossen sind, und sei $C[s]\Downarrow$.
Bilde den Kontext $C' := C[(\lambda x_1, \ldots, x_n.[])x_1 \ldots x_n)$. Dann reduziert $C'[s]$ nach einigen Beta-Reduktionsschritten zu $C[s]$, also gilt $C'[s]\Downarrow$ nach Satz 6.5. Da $C'[s] = C[(\lambda x_1, \ldots, x_n.s)x_1 \ldots x_n]$, erhält man wegen der Voraussetzung auch $C[(\lambda x_1, \ldots, x_n.t)x_1 \ldots x_n]\Downarrow$. Da dies zu $C[t]$ reduziert, erhält man $C[t]\Downarrow$, und damit gilt die Behauptung. □

6.1 Das Kontextlemma

Ein wichtiges Lemma zur kontextuellen Approximation ist die Einschränkung des Kriteriums "für alle Kontexte" auf Reduktionskontexte. Das kann in anderen Kalkülen als gleichwertiger oder überlegener Weg zu den Aussagen wie Korrektheit von Programmtransformationen und Standardisierungslemma verwendet werden. Wir erwähnen das Kontextlemma wegen seiner herausragenden Rolle in anderen Reduktionskalkülen; es läßt sich auch in KFP unabhängig nachweisen, allerdings scheint es nicht der richtige Weg zu sein, die Korrektheit der KFP-Kalkülregeln als Programmtransformationen nachzuweisen. Der Grund könnte darin liegen, dass KFP-Reduktionskontexte offene Terme nicht abschließen können, was in anderen Kalkülen (mit let) möglich ist.
Für das Kontextlemma benötigen wir den technischen Begriff des *Multikontexts*. Das ist eine Verallgemeinerung des Begriffs Kontext auf mehrere Löcher. Dazu kann man z.B. neue Konstanten nehmen, bezeichnet als $\cdot_1, \ldots, \cdot_n$ und Terme bauen, die diese Konstanten enthalten. Wir nehmen zusätzlich an, dass diese Konstanten genau einmal vorkommen, wenn sie erwähnt sind, d.h. $C[\cdot_1, \ldots, \cdot_n]$ ist ein Ausdruck, der jede Konstante $\cdot_i$ genau einmal enthält.

Lemma 6.12. *(Kontext-Lemma) Seien s, t geschlossene Ausdrücke. Wenn für alle (geschlossenen) Reduktionskontexte R: $R[s]\Downarrow \Rightarrow R[t]\Downarrow$ gilt, dann gilt auch $s \leq_c t$.*

Beweis. Der hier gegebene Beweis ist unabhängig von den Beweisen mittels der 1-Relation. Beachte im folgenden, dass $R[s], R[t]$ geschlossene Terme sind.
Wir beweisen die allgemeinere Behauptung:

> Wenn für die geschlossenen Terme s_i, t_i: $\forall i, R : R[s_i]\Downarrow \Rightarrow R[t_i]\Downarrow$ erfüllen, dann gilt für alle Multikontexte $C[\cdot_1, \ldots, \cdot_m]$: Wenn $C[s_1, \ldots, s_n]$, $C[t_1, \ldots, t_n]$ geschlossen sind, dann gilt $C[s_1, \ldots, s_n]\Downarrow \Rightarrow C[t_1, \ldots, t_n]\Downarrow$.

Wie nehmen an, dies sei falsch, Dann gibt es ein Gegenbeispiel, d.h. geschlossene Terme s_i, t_i mit $\forall i, R : R[s_i]\Downarrow \Rightarrow R[t_i]\Downarrow$, aber es gibt einen Multikontext $C[\cdot_1, \ldots, \cdot_n]$, so dass $C[s_1, \ldots, s_n]\Downarrow$, aber $C[t_1, \ldots, t_n]\Uparrow$. Wir wählen den Multikontext so, daß er minimal ist bzgl. der lexikographischen Ordnung mit den Komponenten

1. Die Anzahl der Normalordnungsreduktionen von $C[s_1, \ldots, s_n]$,

2. Die Anzahl der Löcher von C.

Wenn ein Loch von $C[\cdot_1, \ldots, \cdot_n]$ in einem Reduktionskontext ist, d.h., einer der Kontexte $C[\cdot, t_2, \ldots, t_n]$, $C[t_1, \cdot, t_3, \ldots, t_n]$, $\ldots$, $C[t_1, \ldots, t_{n-1}, \cdot]$ ist ein Reduktionskontext, dann nehmen wir der Einfachheit halber an, das erste Loch sei in einem Reduktionskontext. Dann ist $C[\cdot, t_2, \ldots, t_n]$ ein Reduktionskontext. Sei $C' := C[s_1, \cdot_2, \ldots, \cdot_n]$. Da $C'[s_2, \ldots, s_n] \equiv C[s_1, \ldots, s_n]$, haben beide dieselbe Normalordnungsreduktion. Die Anzahl der Löcher von C' ist kleiner als die von C, also gilt $C'[t_2, \ldots, t_n]\Downarrow$. Das bedeutet, dass $C[s_1, t_2, \ldots, t_n]\Downarrow$. Da $C[\cdot, t_2, \ldots, t_n]$ ein Reduktionskontext ist, folgt aus den Bedingungen des Lemmas : $C[t_1, t_2, \ldots, t_n]\Downarrow$, was ein Widerspruch ist.
Wenn keiner der Kontexte $C[\cdot, t_2, \ldots, t_n], C[t_1, \cdot, t_3, \ldots, t_n], \ldots, C[t_1, \ldots, t_{n-1}, \cdot]$ ein Reduktionskontext ist, dann ist entweder $C[s_1, \ldots, s_n]$ eine WHNF, oder die zwei Terme $C[s_1, \ldots, s_n]$ und $C[t_1, \ldots, t_n]$ werden mit derselben Normalordnungsreduktion reduziert. Das liegt daran, dass das Finden des Normalordnungsredex von oben nach unten geschieht und in diesem Falle unabhängig von den Ausdrücken in den Löchern ist.
Falls $C[s_1, \ldots, s_n]$ eine WHNF ist, dann terminiert die Suche nach einem Normalordnungsredex innerhalb des Kontexts C, somit ist auch $C[t_1, \ldots, t_n]$ eine WHNF.
Wir betrachten den Fall dass $C[s_1, \ldots, s_n]$ keine WHNF ist, aber $C[s_1, \ldots, s_n]\Downarrow$. Die Normalordnungsreduktion produziert aus $C[s_1, \ldots, s_n]$ einen Ausdruck $C'[u_1, \ldots, u_m]$ wobei $u_i = s_{\phi(i)}$ für eine Abbildung $\phi : [1..m] \to [1..n]$. Es ist möglich dass $m > n$. Die Anzahl der Normalordnungsschritte von $C'[u_1, \ldots, u_m]$ ist um 1 kleiner als die von $C[s_1, \ldots, s_n]$. Jetzt zeigt die Annahme der Minimalität, dass dann auch $C'[v_1, \ldots, v_m]\Downarrow$ mit $v_i = t_{\phi(i)}$. Da der Ausdruck $C[t_1, t_2, \ldots, t_n]$ in Normalordnung auf $C'[v_1, \ldots, v_m]$ reduziert, gilt auch $C[t_1, t_2, \ldots, t_n]\Downarrow$. □

Beispiel 6.13. Dieses Beispiel zeigt, dass der Beweis des Kontextlemma nicht verallgemeinerbar ist: Nehme den Multikontext $C = (\lambda x.\cdot_1)\cdot_2$. Dann reduziert $C[[x], 1]$ zu $[1]$, aber weder ist eines der Löcher in einem Reduktionskontext, noch kann man die eingesetzten Ausdrücke unverändert im Redukt wiederfinden. D.h. der Beweisschritt, der Normalordnungsreduktion anwendet, und dann das Resultat als neuen Multikontext mit eingesetzten Termen beschreibt, ist in diesem Fall nicht mehr zu begründen.

Übungsaufgabe 6.14. Zeige mit Hilfe des Kontext-Lemmas: Seien $s \to t$ für zwei geschlossene Ausdrücke s, t, wobei die Reduktion unmittelbar ist, d.h. an der Wurzel. Dann gilt $s \sim_c t$.

Leider gibt das Kontext-Lemma keine Information über die kontextuelle Äquivalenz von offenen Ausdrücken. In Zusammenarbeit mit folgender Aussage kann man das Kontextlemma auch in allgemeineren Zusammenhängen verwenden.

Aussage 6.15. *Seien s, t Ausdrücke mit den freien Variablen $x_1, \ldots, x_n$. Dann: $s \leq_c t$ gdw. für alle geschlossenen t_i gilt: $s[t_1/x_1, \ldots, t_n/x_n] \leq_c t[t_1/x_1, \ldots, t_n/x_n]$*

Beweis. Die nichttriviale Richtung der Implikation ist: Wenn für alle geschlossenen t_i gilt: $s[t_1/x_1, \ldots, t_n/x_n] \leq_c t[t_1/x_1, \ldots, t_n/x_n]$, dann gilt $s \leq_c t$.
Induktion nach der Anzahl der freien Variablen in s, t.
Es genügt, für $s' := \lambda x_1, \ldots x_n.s$ und $t' := \lambda x_1, \ldots x_n.t$ zu zeigen, dass $s' \leq_c t'$. Jetzt kann man das Kontextlemma anwenden.
Sei R ein Reduktionskontext. Dann ist das Loch entweder in einem case. Dann sind die Reduktionen von $R[s'], R[t']$ beide mit Typfehler, oder das Loch wird angewendet, d.h. $R[] = R'([]\ r)$, wobei r geschlossen ist. Dann gilt wegen der Induktionsannahme $\lambda x_2, \ldots, x_n.s[r/x_1] \leq_c \lambda x_2, \ldots, x_n.t[r/x_1]$. D.h. $R[s'] \sim_c R'([s']\ r) \sim_c R'(\lambda x_2, \ldots, x_n.s[r/x_1])$. Somit gilt: $R[s']\Downarrow \Rightarrow R[t']\Downarrow$. Das Kontextlemma zeigt jetzt $s' \leq_c t'$, und somit auch $s \leq_c t$. $\square$

6.2 Programmtransformationen

Programmtransformationen sind zunächst mal alle Veränderungen eines Programms P resultierend in einem Programm P'. Natürlich soll diese Transformation nicht das Programm verfälschen, z.B. aus einem Programm, das Zahlen testet, ob sie gerade sind, ein Programm machen, das Zahlen testet, ob sie ungerade sind. Dies wird durch den Begriff der kontextuellen Gleichheit erfaßt.

Definition 6.16. *Eine* Programmtransformation *ist eine Relation R auf Ausdrücken. Man sagt zu $e_1\ R\ e_2$, dass R den Ausdruck e_1 zu einem Ausdruck e_2* transformiert.
Die Programmtransformation R ist korrekt, *wenn aus $e_1\ R\ e_2$ stets $e_1 \sim_c e_2$ folgt.*
Die Programmtransformation R ist modular, *wenn für jeden Kontext C aus $e_1\ R\ e_2$ auch $C[e_1]\ R\ C[e_2]$ folgt.*

Modularität einer Programmtransformation bedeutet, dass man sie auf Unterterme anwenden kann ohne den Rest des Programms zu kennen.
Beachte, dass die Korrektheit von Programmtransformationen zunächst nur bedeutet, das diese während des Kompilierens verwendet werden dürfen, während des Programmlaufs, d.h. in eine Normalordnungsreduktion eingeschoben nur dann, wenn man darauf achtet, dass sich die Terminierungseigenschaften nicht ändern. Eine veränderte Normalordnung könnte auch eine gewünschte Verbesserung der Auswertung sein, aber dazu sind andere Betrachtungen notwendig: Man muss dazu verschiedene Reduktionsstrategien vergleichen, was wir an anderer Stelle machen werden.

Beispiel 6.17. Ein triviales Beispiel für eine veränderte Normalordnungsreduktion, die aus korrekten Programmtransformation während der Normalordnungsreduktion besteht, aber ein falsches Terminierungsverhalten hat, ist: Mache einen Normalordnungsreduktionsschritt, und dann mache diesen wieder rückgängig. Diese "Reduktion" ändert die Terme nicht, kann also i.a. nicht terminieren.

Korollar 6.18. *Aus dem Satz 6.5 oben folgt, dass die Beta- und die case-Reduktion an beliebigen Stellen eines Programms während des Kompilierens als Programmtransformationen verwendet werden dürfen. Das gilt auch für die umgekehrten Regeln. D.h. beta-Expansion und case-Expansion.*

Die meisten KFP-Programmtransformationen sind eine Kombination von Kalkülregeln. Es gibt auch solche, die keine Kombination von Kalkülregeln sind. (z.B. Assoziativität von append $(++)$)
Die sogenannte Lambda-lifting Transformation ist im ungetypten Lambda-kalkül, d.h. ohne case und Konstruktoren, eine bekannte Methode, um Superkombinatorprogramme herzustellen.

Definition 6.19. *Sei $\lambda x.e$ ein Ausdruck, so dass e eine freie Variable y enthält. Dann ist die Lambda-lifting Transformation:*
$\lambda x.e \xrightarrow{ll} (\lambda z.\lambda x.e[z/y])\ y$, *wobei z eine neue Variable ist.*

Aussage 6.20. *Lambda-Lifting ist in KFP eine korrekte Programmtransformation.*

Beweis. Das folgt daraus, dass sich $(\lambda z.\lambda x.e[z/y])\ y$ mittels beta-Reduktion wieder zu $\lambda x.(e[z/y])[y/z] = \lambda x.e$ reduzieren läßt, und dass Beta eine korrekte Programmtransformation ist. □

Die sogenannte η-Transformation ist ebenfalls im ungetypten Lambda-kalkül bekannt. Man kann diese zum Lambda-Kalkül hinzunehmen oder nicht, was natürlich verschiedene Gleichheiten ergibt.

Definition 6.21. *Sei $\lambda x.(e\ x)$ ein Ausdruck, so dass e die Variable x nicht enthält. Dann ist die η-Transformation:*

$$\lambda x.(e\ x) \quad \xrightarrow{\eta} \quad e$$

Lemma 6.22. *Die η-Transformation ist nicht korrekt in KFP.*

Beweis Dazu genügt es einen Ausdruck anzugeben, für den das falsch ist. Es gibt Ausdrücke, die nicht terminieren. z.B. $bot := (\lambda x.(x\ x))\ (\lambda y.(y\ y))$. Dann hat $(\lambda x.bot\ x)$ im leeren Kontext eine WHNF, aber es gilt $\lambda x.bot\ x \xrightarrow{\eta} bot$, und bot hat keine WHNF. □

Die η-Reduktion ist allerdings in vielen Fällen korrekt. U.a. wenn der eta-reduzierte Term auf etwas angewendet wird.

6.3 Kleinste und Größte Elemente bzgl. $\leq_c$

Was wir auch zeigen können, ist die Äquivalenz aller nichtterminierenden Ausdrücke und ebenso die Äquivalenz von nichtterminierenden Ausdrücken mit nicht getypten Ausdrücken.

Offene Ausdrücke, die ungetypt sind, oder die eine nichtterminierende Normalordnungsreduktion haben, behalten diese Eigenschaft auch nach Einsetzung von Termen für die freien Variablen:

1. Sei $t \xrightarrow{no} t_0$, so dass t_0 direkt ungetypt. (wir haben das dynamisch ungetypt genannt). Dann ist für jede Einsetzung ρ auch der Ausdruck $t\ \rho$ dynamisch ungetypt.
   Das liegt daran, dass die Normalordnungsreduktion freie Variablen nicht unmittelbar anfaßt, sondern nur mitschleppt.
2. Sei $t \xrightarrow{\infty,no}$. Dann gilt für jede Einsetzung ρ auch $t\ \rho \xrightarrow{\infty,no}$. Auch hier ist die Normalordnungsreduktion die gleiche.

Lemma 6.23. *Sei s ein geschlossener Ausdruck mit $s\Uparrow$ und t ein weiterer geschlossener Ausdruck. Dann gilt $s \leq_c t$.*

Beweis. Sei R ein Reduktionskontext. Jede Normalordnungsreduktion von $R[s]$ muß zuerst s in WHNF reduzieren, was nach Voraussetzung nicht geht. Damit gilt $R[s]\Downarrow \Rightarrow R[t]\Downarrow$ und somit folgt aus dem Kontextlemma, dass $s \leq_c t$ □

Lemma 6.24. *Sei s ein (evtl. offener) Ausdruck, so dass für alle Einsetzungen ρ: $(s\ \rho)\Uparrow$, und sei t ein weiterer Ausdruck. Dann gilt $s \leq_c t$.*

Beweis. Das Lemma oben zeigt, dass für alle Einsetzungen ρ, die alle freien Variablen von s, t mit geschlossenen Ausdrücken ersetzen, dass $s\rho\Uparrow$, und damit $s\rho \leq_c t\rho$. Lemma 6.15 zeigt dann auch $s \leq_c t$. □

Damit gibt es ein kleinstes Element in der $\leq_c$-Ordnung in der Menge der geschlossenen Ausdrücke, nämlich *bot*, auch als $\perp$ geschrieben.

Aussage 6.25. *Seien s, t zwei (evtl. offene) Ausdrücke, die entweder eine nicht terminierende Normalordnungsreduktion haben oder direkt nicht wohlgetypt sind. Dann gilt $s \sim_c t$.*

Beweis. Folgt aus obigem Lemma, da wegen $s\Uparrow$ und $t\Uparrow$ sowohl $s \leq_c t$ als auch $t \leq_c s$ gilt. □

Beachte, dass es auch offene Terme gibt, die eine unendliche Normalordnungsreduktion haben: Z.B. `bot x`. Für diese gilt ebenfalls `bot x` $\sim_c$ `bot`.

Aussage 6.26. *Wenn es mindestens einen Konstruktor gibt, dann gibt es kein größtes Element in der $\leq_c$-Ordnung*

Beweis. Sei T ein Typ mit Konstruktor c der Stelligkeit n: Ein größtes Element s muß eine WHNF haben. Dafür gibt es zwei Möglichkeiten:

– $s = \lambda x\ .\ s'$. In dem Fall ist $\mathtt{c}\ \underbrace{\mathtt{c} \ldots \mathtt{c}}_{n} \not\leq_c s$: Nehme den Kontext $\mathtt{case_T}\ []\ ((c\ x_1 \ldots x_n) \to c) \ldots$.

– $s = d\ s_1 \dots s_m$ wobei d ein Konstruktor ist. Sei m' die Stelligkeit des Konstruktors, und sei $k = m' - m$. In dem Fall ist $\lambda x_1, \dots, x_{k+1}\ .\ c \not\leq_c s$: Nehme den Kontext $C[] := (\cdot\ \underbrace{\mathtt{c} \dots \mathtt{c}}_{k+1})$. Für s ergibt dies einen ungetypten Ausdruck $C[s]$, aber $C[\lambda x_1, \dots, x_{k+1}\ .\ \mathtt{c}]$ reduziert zu $\mathtt{c}$.

□

Diese Aussage zeigt, dass die Simulation von KFP im Lambda-Kalkül nicht 1-1 ist, denn der Lambda-kalkül mit der kontextuellen Ordnung wie in "The lazy Lambda-Calculus" von Abramsky beschrieben, hat ein eindeutiges größtes Element (modulo der kontextuellen Ordnung): $(Y\ K)$[1], das ist ein Lambdaausdruck, der beliebig viele Argumente schluckt, ohne sich zu verändern: $(Y\ K)\ t \sim_c (Y\ K)$, wobei $\sim_c$ hier die kontextuelle Ordnung im lazy Lambda-Kalkül ist.

6.4 Auswertungsreihenfolgen

Aus dem Satz über die Korrektheit der Anwendbarkeit von Reduktionen an beliebigen Stellen in einem Ausdruck folgt, dass man Ausdrücke in einem Programm in beliebiger Reihenfolge auswerten darf. Allerdings gilt das nur, wenn die Auswertungen auch terminieren.
Folgende Normalform für strikte Auswertung entspricht der Auswertung der Argumente einer Anwendung vor dem Einsetzen der Argument.

Definition 6.27. *Eine* strikte Normalform *ist definiert als:*

- *Eine Abstraktion ist eine strikte Normalform.*
- *Wenn $s_1, \dots, s_n$ strikte Normalformen sind, dann auch $c\ s_1 \dots s_n$ für einen Konstruktor c mit $ar(c) \geq n$.*

Zwei Strategien bzw. Methoden der Reduktion haben eigene Namen:

Normalordnung (normale Reihenfolge, normal-order, nicht-strikte Auswertung), die wir bereits kennen.

Anwendungsordnung (applikative Reihenfolge, applikative Ordnung, strikte Auswertung). Hier wird verlangt, dass man die Beta-Reduktion nur verwenden darf, wenn das Argument ausgewertet ist, d.h., eine strikte Normalform ist.

Es gibt auch andere Varianten der Auswertung, in denen Normalform durch etwas anderes (z.B. HNF) ersetzt ist.
Aus Satz 6.9 folgt, dass die verschiedenen Methoden die gleichen Resultate liefern, wenn die Auswertung mit Konstruktortermen terminiert.
Allerdings kann das Verhalten bzgl. Terminierung verschieden sein:

[1] Y ist der Fixpunktkombinator wie in Abschnitt 7.2 beschrieben und K ist definiert als $K\ x\ y = x$

Beispiel 6.28. Sei $K = \lambda x.(\lambda y\ .\ x)$.
Dann wertet die Anwendungsordnung den Ausdruck K `Nil` bot so aus, dass zuerst versucht wird, `bot` in Normalform zu bringen. Das gelingt nicht, also terminiert das nicht.
Die Normalordnung wertet den Ausdruck K `Nil` bot so aus, dass zuerst der Ausdruck $((\lambda y\ .\ \texttt{Nil})\ bot)$ entsteht. Danach wird bot eingesetzt, und der Ausdruck reduziert zu `Nil`. D.h. in diesem Fall terminiert die Auswertung.

Es gilt folgender Satz (Church,Rosser) der in der Literatur unter Existenz einer Standardreduktion für den Lambda-Kalkül bekannt ist. Wir formulieren ihn leicht verallgemeinert.

Satz 6.29. *Sei s ein geschlossener Ausdruck. Wenn es einen Ausdruck t in WHNF gibt mit $s \sim_c t$, dann gilt $s\Downarrow$.*

Beweis. Unsere Definitionen sind auf diesen Fall zugeschnitten: Nehme den leeren Kontext. Dann folgt aus $s \sim_c t$ und $t\Downarrow$ dass auch $s\Downarrow$ □

Insbesondere heißt das:

Lemma 6.30. *Wenn für einen Term t die Reduktion in Anwendungsordnung terminiert, dann auch die Reduktion in Normalordnung*

Beweis. Mit Induktion über die Tiefe der entstehenden strikten Normalform. □

Beispiel 6.31. Zu verschiedenen Reihenfolgen der Auswertung von Ausdrücken. Wertet man $(quadrat(3 * 3))$ aus, so kann man das auf zwei Weisen machen:

$$(quadrat(3\ *\ 3)) \to quadrat\ 9 \to 81$$

oder

$$(quadrat(3 * 3)) \to ((3 * 3) * (3 * 3)) \to (9 * (3 * 3)) \to (9 * 9) \to 81$$

Bemerkung: Programmiersprachen, die die Anwendungsordnung als Auswertungsstrategie haben, nennt man *strikte FPS*, solche die die Normalordnung benutzen, nennt man auch *nicht-strikte FPS*. Der Begriff kommt eigentlich aus der denotationalen Semantik der Programmiersprachen, wird aber auch für die passende operationale Auswertung verwendet.

6.5 Relation zur Konfluenz

Im Lambdakalkül verwendet man den Begriff der $\alpha\beta$-Gleichheit, um Gleichheit von Ausdrücken zu spezifizieren: Wenn man diesen nach KFP überträgt, dann ergibt sich:

> zwei Ausdrücke s, t sind $\alpha\beta$-gleich ($s =_{\alpha\beta} t$ wenn es eine Folge von (Beta und Case-) Reduktionen (vorwärts oder rückwärts) und Umbenennungen gibt, die s in t transformiert.

Es gilt: $s =_{\alpha\beta} t \Rightarrow s =_c t$.

Die *Konfluenz* ist eine Eigenschaft, die syntaktisch analog zur Standardreduktion ist:

Wenn $s \xrightarrow{*} s_1, s \xrightarrow{*} s_2$, dann existiert $t_1 =_\alpha t_2$ mit: $s_1 \xrightarrow{*} t_1, s_2 \xrightarrow{*} t_2$.

Diese Konfluenzeigenschaft gilt ebenfalls in KFP für die Reduktion. Sie läßt sich ganz analog mit den Mitteln zeigen, die wir bereitgestellt haben: mit der 1-Relation. Allerdings ist die Konfluenz nicht die wesentliche Eigenschaft. Wesentlich ist, dass alle Reduktionen korrekte Programmtransformationen sind. Aus der Konfluenz selbst folgt diese Eigenschaft nicht, da die Konfluenz nichts darüber aussagt, von welcher Art die schließenden Reduktionen sind (normal order j/n).

Beispiel 6.32. Es gibt Ausdrücke, die gleich sind bzgl. $=_{\alpha\beta}$ aber nicht gleich bzgl. $=_c$:
Das Beispiel ist abgeleitet aus den Haskell-Definitionen

```
repST x = S (repST True)
repSF x = S (repSF False)
```

Die Definitionen sind:

```
Y  =  \f. (\ x. f (x x)) (\x. f (x x))
A  =  Y (\repS . \x. (S (repS True ))) Nil
B  =  Y (\repS . \x. (S (repS False))) Nil
```

Durch langwieriges Auswerten von Hand sieht man, dass ein "unendlicher" Ausdruck der Form `S (S (S ...)))` erzeugt wird. Hieraus folgt $A =_c B$. In diesem Skript wurden die Mittel dazu noch nicht erläutert.
Aber man kann die Ausdrücke nicht ineinander überführen mit $\alpha\beta$-Reduktion. Ein Mittel, um das letztere zu zeigen, ist die Konfluenz, bzw. in Verallgemeinerung die Church-Rosser-Eigenschaft, dass Ausdrücke $s =_{\alpha\beta} t$ einen gemeinsamen Nachfolger (modulo α) haben. Denn A und alle mit Reduktion daraus herleitbaren Ausdrücke haben im Innern `True` , aber B und alle mit Reduktion daraus herleitbaren Ausdrücke haben im Innern `False` .

7 Zusammenhang zwischen KFP und Superkombinatoren

Wir untersuchen in diesem Kapitel den Zusammenhang zwischen KFP und KFP+. Zur Erinnerung an die Definitionen der Reduktion siehe Definition von KFP. In diesem Kapitel seien in KFP+ keine Abstraktionen erlaubt.
Der Schritt von Superkombinatoren nach KFP ist nicht so groß, trotzdem sollte man sich darüber klar werden, ob alle KFP-Ausdrücke übersetzt werden können und ob die Übersetzung korrekt ist.
Neben trivialen syntaktischen Unterschieden gibt es folgende Unterschiede:

- In KFP kann man β-Reduktion bereits bei einem Argument anwenden, in KFP+ die SK-Beta-Reduktion erst, wenn ausreichend viele Argumente vorhanden sind.
- In KFP+ gibt es allgemeine Rekursion. In KFP gibt es keine (direkte) Rekursion.
- in KFP+ gibt es keine Lambda-Ausdrücke, in KFP gibt es Lambda-Ausdrücke.

Wir geben zwei Übersetzungen σ und τ an:

- $\sigma : KFP \to KFP+$ wobei lambda-freie Superkombinatorausdrücke entstehen.
- $\tau : KFP+ \to KFP$.

Im folgenden steht die Korrektheit dieser Transformationen im Zentrum. Dazu diskutieren wir auch die Kriterien und die Mittel zum Nachweis der Korrektheit. Um die Betrachtungen zur Übersetzungsfunktion zu vereinfachen, nehmen wir an, dass die Typen und Konstruktoren im KFP und im KFP+-Programm jeweils dieselben sind.

7.1 Von KFP nach KFP+: Lambda-Lifting

Erster Schritt der Übersetzung ist die Elimination von lokalen Abstraktionen, die lokal freie Variablen enthalten. Dies wäre nicht direkt in lambda-freies KFP+ übersetzbar.

Definition 7.1. *Sprechweisen in der Sprache KFP (im Lambda-kalkül):*

- *Ein* Kombinator *ist ein geschlossener Ausdruck.*
- *Eine* maximale Abstraktion t' *in einem Ausdruck* t *ist entweder* t *selbst, wenn* t *eine Abstraktion ist, oder ein Unterausdruck, dessen direkter Oberterm keine Abstraktion ist.*
- *Ein* Superkombinator t *ist ein geschlossener KFP-Ausdruck* e*, so dass alle maximalen Abstraktionen in* e*, die echte Unterterme von* t *sind, ebenfalls Superkombinatoren sind.*
- *Ein Ausdruck, dessen sämtliche maximale Abstraktionen Superkombinatoren sind, heißt* Superkombinatorausdruck.

Kombinatoren und Superkombinatoren kann man als eigenständige Objekte ansehen.

Beispiel 7.2.

- $\lambda x_1, x_2, \ldots x_n \;.\; x_i$ ist ein Superkombinator.
- $\lambda x \;.\; (x\ \lambda y \;.\; x)$ ist kein Superkombinator, da die maximale Abstraktion $\lambda y \;.\; x$ eine freie Variable enthält.
- $(\lambda x.\lambda y.(y\ (\lambda x.x)))$ ist ein Superkombinator.
- `case Nil` $((x : xs) \to x)$ $($`Nil` $\to$ `Nil`$)$ ist ebenfalls ein Superkombinator. Allerdings einer, der keine Argumente hat.

Die Regel *Lambda-lifting* verwenden wir jetzt, um aus jedem geschlossenen KFP-Ausdruck einen kontextuell äquivalenten KFP-Superkombinatorausdruck zu machen.

> Sei $t \equiv C[t']$ wobei t' eine echte maximale Abstraktion in t ist, und jeder echte Unterterm von t', der eine maximale Abstraktion ist, ist ein Superkombinator, und y ist frei in t'. Dann:
>
> $$C[t'] \xrightarrow{ll} C[((\lambda z \ . \ (t'[z/y]))\ y)]$$
>
> wobei z eine neue Variable ist.

Offenbar ist diese Regel korrekt, was direkt aus den Sätzen zur kontextuellen Gleichheit in KFP folgt, da sich diese Transformation mit Beta-Reduktion wieder rückgängig machen läßt.
Außerdem terminiert diese Regel und erzeugt einen Superkombinatorausdruck:

Lemma 7.3. *Jeder geschlossene KFP-Ausdruck kann von endlich vielen* $\xrightarrow{ll}$*-Transformationen in einen KFP-Superkombinatorausdruck transformiert werden.*

Beweis. Die Regel ist immer anwendbar, wenn t kein Superkombinatorausdruck ist. Also zeigen wir, dass die fortgesetzte Regelanwendung von $\xrightarrow{ll}$ terminiert. Am einfachsten ist das folgendermaßen einzusehen: Jeder Superkombinator im Ausdruck wird durch einen Namen abgekürzt. Das ändert nichts an der Ausführbarkeit und Ausführung der Regel. Eine einfache Anwendung auf t' vermindert die Anzahl der freien Variablen in t', und wenn der Ausdruck ein Superkombinator ist, dann kann er abgekürzt werden. Insgesamt wird danach die Anzahl der λ's weniger, auch wenn man die vom Lambda-Lifting neu eingeführten mitzählt. Das terminiert, da nach endlich vielen Anwendungen keine Abstraktionen mehr vorhanden sind, alle sind in Abkürzungen ausgelagert. □

Superkombinatorübersetzung KPF → KFP+

Die am einfachsten zu überschauende Vorgehensweise bei der Übersetzung ist das Hinzufügen von Superkombinatordefinitionen zu einem KFP-Ausdruck in der Form von Abkürzungen. D.h. wir nehmen wie oben an, dass es eine Menge von Abkürzungen der Form

$$name := E$$

gibt, wobei E ein KFP-Superkombinator ist, und E aber möglicherweise auch Namen enthält. Wenn man das noch syntaktisch etwas umschreibt in

$$name\ x_1 \ldots x_n := E'$$

wobei $E \equiv \lambda x_1 \ldots x_n \ . \ E'$, dann hat man KFP+-Definitionen. Die Stelligkeit eines Superkombinators ist die Anzahl der im Lambda-Präfix des Superkombinators gebundenen Variablen.

Beispiel 7.4. Man muss in KFP+ für die Reduktion (SK-Beta) statt (Beta) verwenden, denn die KFP-Beta-Reduktion würde die Struktur zerstören: Sei `P x y z = z` eine Definition und `(P 1 2 3)` ein Ausdruck. Die Beta-Reduktion angewendet auf `((\x,y,z . z) 1 2 3)` erzeugt dann als Ausdruck: `(\y,z . z) 2 3`, aber der Superkombinator `(\y,z . z)` ist nicht in der Liste der Definitionen.

Beschränkt man sich auf KFP-Normalordnungsreduktionen von geschlossenen Ausdrücken, dann stellt sich heraus, dass in der KFP-Normalordnungs-Reduktion von Superkombinatorausdrücken sich Reduktionen so zusammenfassen lassen, dass man keine neuen Definitionen braucht. Die Anzahl der Reduktionen die man zusammenfassen muß, entspricht genau der Anzahl der Argumente des Superkombinators, der aktuell reduziert wird. Einzige Ausnahme sind die Reduktionen kurz vor Erreichen der KFP-WHNF oder kurz vor der Erkennung eines Typfehlers.
Es gilt:

- Reduziert man einen geschlossenen KFP-Ausdruck in KFP-Normalordnung, dann werden sowohl in der Beta als auch in der Case-Reduktion nur geschlossene Ausdrücke für Variablen eingesetzt.
- Wenn man in KFP+ einen KFP+-Normalordnungsredex hat, der in eine Applikation $F\ t_1 \ldots t_n$ eingebettet ist, wobei F ein Superkombinatorname ist, der als $F\ x_1 \ldots x_m = r$ definiert ist und $m \leq n$, dann ergeben m Schritte der KFP-Normalordnungsreduktion:

  $$r[t_1/x_1, \ldots, t_m/x_m]\ t_{m+1} \ldots t_n$$

  Dieser Ausdruck ist wieder ein geschlossener Superkombinatorausdruck, da die t_i geschlossene Superkombinatorausdrücke sind. Diese m Reduktionen entsprechen genau einem SK-Beta wie er in KFP+ definiert wurde.

  Ist der KFP-Normalordnungsredex in eine Applikation $F\ t_1 \ldots t_n$ eingebettet, wobei die Stelligkeit von F gilt $m > n$, und der Oberterm der Applikation keine Applikation ist, d.h. ein `case` oder die Applikation ist die Wurzel, dann ergibt sich entweder nach m Reduktionen ein dynamischer Typfehler, oder eine WHNF.
- Eine `case`-Reduktion in KFP-Normalordnung ist dasselbe wie eine KFP+-Normalordnungsreduktion und aus dem gleichen Grunde unproblematisch und ergibt einen geschlossenen Superkombinatorausdruck.

Folgendes Diagramm zeigt die wesentliche Eigenschaft der Übersetzung σ: Wenn t nach $\sigma(t)$ übersetzt wird, und man einen Normalordnungsreduktionsschritt von $\sigma(t)$ aus machen kann, dann existiert ein t' mit den Eigenschaften, wie im Diagramm angegeben

$$\begin{tikzcd} t \arrow[r] \arrow[d, "no,*"] & \sigma t \arrow[d, "no"] \\ \exists t' \arrow[r] & \sigma t' \end{tikzcd}$$

Aussage 7.5. *Ist ein geschlossener Ausdruck t in KFP+ gegeben, so dass t keine Abstraktionen enthält, und alle notwendigen Superkombinatordefinitionen nicht rekursiv sind, dann ist die KFP-Normalordnungsreduktion als Superkombinatorreduktion durchführbar, bis entweder ein Typfehler auftritt, oder ein Superkombinator zuwenig Argumente hat. Im letzten Fall ist der Ausdruck in KFP+-WHNF.*

Beweis. Wenn die Definitionen nicht rekursiv sind, dann kann man die Definitionen wie Abkürzungen behandeln und einfach expandieren. Danach sind alle Namen im Ausdruck ersetzt und es ergibt sich ein geschlossener KFP-Ausdruck. Wie oben erwähnt, wirkt die KFP+-Normalordnung wie eine Zusammenfassung von KFP-Normalordnungsschritten. □

Ein KFP-Superkombinatorausdruck ist problemlos nach KFP+ zu übersetzen, wobei alle Definitionen und der Ausdruck lambdafrei sind:

Definition 7.6. *σ ist definiert auf einem KFP-Superkombinatorausdruck und erzeugt einen lambdafreien KFP+-Ausdruck und nichtrekursive KFP+-Superkombinatordefinitionen; d.h. ein lambda-freies, nicht-rekursives KFP+-Programm. Die KFP+-Definitionen seien genau die Abkürzungen, die sich wie beim Lambdaliften ergeben.*
Die Übersetzung σ besteht dann genau aus dem Erzeugen der Abkürzungen für alle Superkombinatoren:

Die Aussage 7.5 ergibt, dass für alle Ausdrücke in KFP ihr Bild in KFP+ folgende Eigenschaft hat:

Aussage 7.7. *Sei t ein geschlossener Ausdruck. Dann ist $t\Downarrow \Leftrightarrow \sigma(t)\Downarrow_{KFP+}$*

D.h Diese Übersetzung σ ist korrekt unter dem Kriterium der Erhaltung der Terminierung.
Die Erhaltung der kontextuellen Approximationsordnung (und damit auch der Äquivalenz) von Ausdrücken bei der Übersetzung σ kann man mit einer leichten Einschränkung schon formulieren: Es gilt:

Aussage 7.8. *Seien s, t geschlossene Ausdrücke. Dann gilt*

$$s \sim_c t$$
$$\Leftrightarrow$$
$$\forall C: \text{ wenn } C \text{ ein Kontext ohne rekursive Definitionen ist und}$$
$$C[\sigma(s)], C[\sigma(t)] \text{ geschlossen sind, dann } C[\sigma(s)]\Downarrow_{KFP+} \Leftrightarrow C[\sigma(t)]\Downarrow_{KFP+}$$

Beweis. Das gilt, da die KFP+-Kontexte, die keine rekursiven Superkombinatoren enthalten, als KFP-Kontexte leicht umformbar sind. Die Reduktionen lassen sich mit Aussage 7.7 hin und her übersetzen. □

In dieser Aussage fehlen in den KFP-Kontexten die rekursiven Definitionen. Das Kriterium, das erfüllt sein sollte, ist:

$$s \leq_c t \Leftrightarrow \sigma(s) \leq_{c,KFP+} \sigma(t)$$

Dazu die Definition von $\leq_{c,KFP+}$:

Definition 7.9. *Zunächst mal braucht man ein KFP+-Programm P, das aus den aktuellen Superkombinatordefinitionen besteht, so dass s, t Ausdrücke sind, die mit Superkombinatoren aus P formulierbar sind. Ein Oberprogramm-Kontext $P'[]$ von P ist eine Menge von Superkombinatordefinitionen, die zumindest die aus P und evtl. noch mehr enthält, wobei das Loch $[]$ in einem neuen Rumpf enthalten ist. Dann definiert man:*
$s \leq_{c,KFP+} t$ gdw. für alle Oberprogramme P' von P und alle Oberprogramm-Kontexte $P'[]$: $main_{P'[s]}\Downarrow \Rightarrow main_{P'[t]}\Downarrow$.

Der Grund zur Definition von "Programmkontexten" ist: Wenn s, t offen sind, dann benötigt man Programmkontexte, d.h. man muss die Ausdrücke auch in eine Definition einsetzen dürfen, da man keine Lambdas in KFP+ hat, und sonst Ausdrücke mit freien Variablen nicht testen kann.

7.2 Übersetzung KFP+ nach KFP

Für die Übersetzung in die andere Richtung, insbesondere von rekursiven Superkombinatoren, benötigen wir etwas mehr Aufwand, da man Rekursion nicht direkt nach KFP übersetzen kann.
Das Stichwort heisst: *Fixpunkte und Fixpunktkombinatoren.*
Die Auflösung der Rekursion ist möglich mit einem Superkombinator Y, der folgendes leistet:
Für alle Abstraktionen F gilt in KFP: $Y\ F \sim_c F\ (Y\ F)$ D.h. $(Y\ F)$ ist ein Fixpunkt von F. Da Y somit angewendet auf F einen Fixpunkt von F konstruiert, nennt man Y auch einen *Fixpunktkombinator.*
Solch einen Kombinator gibt es:

$$Y = \lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$$

Übungsaufgabe 7.10. Definiere Y mittels lambdafreier Superkombinatoren.

Wir reduzieren $(Y\ F)$:

$$\begin{aligned}(Y\ F) &\rightarrow (\lambda x.F\ (x\ x))(\lambda x.F\ (x\ x))\\ &\rightarrow F\ (x\ x)[x/(\lambda x.F\ (x\ x))]\\ &= F((\lambda x.F\ (x\ x))(\lambda x.F(x\ x))\end{aligned}$$

Satz 7.11. *Für alle Abstraktionen F gilt $Y\ F \sim_c F\ (Y\ F)$.*

Beweis. Folgt aus den Sätzen über kontextuelle Gleichheit der Reduktion in KFP □

Diese Äquivalenz kann man ausnutzen, um Rekursion aus den KFP+-Definitionen von Superkombinatoren zu eliminieren.

Wir machen das zuerst für einfache Rekursion:
Sei `H~x_1 \ldots x_n = e` eine rekursive Definition in KFP+. Wir nehmen an, dass e nur Vorkommen von H enthält, aber keine anderen Superkombinatoren. Dann erzeuge den Ausdruck

$$G = \lambda f.\lambda x_1 \ldots \lambda x_n.e[f/H]$$

Danach definieren wir $F = (\lambda x.G(x\ x))\ (\lambda x.G(x\ x))$. Das entspricht $F = Y\ G$ nach einer Reduktion. Dies ist ein Ausdruck in KFP ohne Vorkommen eines Superkombinatornamens.
Wir zeigen, dass F zu H äquivalent ist in folgendem Sinn: Die Terminierung ändert sich nicht, wenn man vom KFP+-Ausdruck mit H zu dem KFP-Ausdruck mit F übergeht.

Definition 7.12. *Sei die vereinfachte Funktion τ die Übersetzung, die in jedem KFP+-Ausdruck die Vorkommen von H durch F ersetzt.*

Aussage 7.13. *Sei in KFP+ nur eine einzige einfache rekursive Definition wie oben. Sei t ein geschlossener KFP+-Ausdruck, und sei τ wie oben definiert. Dann ist $t\Downarrow_{KFP+} \Leftrightarrow \tau(t)\Downarrow$*

Beweis. Wir haben schon gesehen, dass der Unterschied zwischen der KFP-Normalordnungsreduktion und der KFP+-Normalordnungsreduktion nicht wesentlich ist. Man kann zeigen, dass sich Normalordnungsreduktion mit τ überträgt. Der einzig interessante Fall, der einen Unterschied machen könnte, tritt ein, wenn der Normalordnungsredex gerade $H\ a_1 \ldots a_n$ ist. Dann reduziert dieser Term auf der KFP+-Seite zu $e[a_1/x_1, \ldots a_n/x_n]$. Auf der KFP-Seite ergibt sich in diesem Fall:

$$\begin{array}{lll} F\ a_1 \ldots a_n & \rightarrow & (\lambda x.G\ (x\ x))(\lambda x.G\ (x\ x))\ a_1 \ldots a_n \\ & \rightarrow_{no} & G((\lambda x.G\ (x\ x))(\lambda x.G(x\ x)))\ a_1 \ldots a_n \\ & \xrightarrow{*}_{no} & e'[(\lambda x.G\ (x\ x))(\lambda x.G\ (x\ x))/H; a_1/x_1, \ldots, a_n/x_n] \end{array}$$

Das ist gerade die Übersetzung $\tau(e[a_1/x_1, \ldots a_n/x_n])$. Da sich WHNF entsprechen (bis auf wenige KFP-Reduktionen, ergibt sich mit Induktion nach der Anzahl der KFP+-Normalordnungsreduktionen die Behauptung. □

Der Fall, dass die Rekursion verschränkt ist, bzw. dass die Verwendung der Superkombinatornamen nicht eingeschränkt ist, ist einfach zu modellieren, da es Konstruktoren gibt:

Wenn in einem KFP+-Programm die (rekursiven) Superkombinatoren $S_1, \ldots, S_n$ definiert sind, dann fügen wir einen Typ T hinzu mit einem n-stelligen Konstruktor c und zugehörigem $\texttt{case}_T$.
Die Übersetzung kann dann mit einer einzigen rekursiven Funktion F gemacht werden, wobei F so definert ist, dass es eine Fallunterscheidung über das erste (ganzzahlige) Argument macht, so dass $(F\ i)$ gerade S_i implementiert, Die Vorkommen von S_i werden in allen Definitionen werden durch $F\ i$ ersetzt.
Die Problematik verschiedener Stelligkeit kann leicht umgangen werden, indem man F so definiert, das die Stelligkeit 1+ das Maximum der Stelligkeiten der S_i ist. Das S_i wird jeweils durch $F\ i\ d\ \ldots\ d$ ersetzt, wobei man das Dummyargument d gerade so oft hinzufügt, dass die Stelligkeit stimmt.
Damit haben wir alle Funktionen in eine einzige rekursive Funktion kodiert.
Diese Methode ist etwas zu direkt, da sie einerseits neue Konstruktoren einführt, und andererseits beim Hinzufügen von unabhängig definierten Funktionen die Übersetzung ändert.

Beispiel 7.14. Sei S_1 einstellig und S_2 zweistellig. Dann kann man S_1 jeweils ersetzen durch $F\ 1\ d$ und S_2 durch $F\ 2$. Die Stelligkeiten sind dann richtig.

Beispiel 7.15. Angenommen, wir haben zwei verschränkt rekursiv definierte Funktionen f, g.

```
f x = f (f x) (g x)
g x = g x (f x)
```

Dann definieren wir eine Funktion G:

```
G  = \F  i x . case_int i (1 ->  (F  1 (F  1 x) (F  2 x))
                          (2 ->  (F  2 x (F  1 x)))
```

Bildet man jetzt $Y\ G$, und ersetzt alle Vorkommen von f durch `Y G 1` und alle Vorkommen von g durch `Y G 2`, dann hat man die Rekursion eliminiert.
Denn `g a` wird ersetzt durch `Y G 2 a`. Dies wird durch Reduktion zu `G ((\x. G(x x))((\x. G(x x)))) 2 a` und wenn wir `((\x. G(x x))((\x. G(x x))))` mit `GG` abkürzen, dann wird dies wird Reduktion zu `(GG 2 x (GG  1 x)))`.
Man sieht auch, dass es nicht nötig ist, die Funktionen unter einem Konstruktor zusammenzufassen. Es genügt eine Fallunterscheidung.

Definition 7.16. *Die Übersetzung $\tau : KFP+ \to KFP$ sei so definiert, dass Superkombinatoren, die nicht rekursiv sind, direkt nach KFP übersetzt werden. Rekursive Superkombinatoren werden in Gruppen zusammengefaßt, so dass sich Funktionen aus verschiedenen Gruppen nicht aufrufen. Danach kann man die Übersetzung gruppenweise durchführen. Dies garantiert, dass man neue Definitionen zu einem Programm hinzufügen kann, ohne dass sich die Übersetzung bereits vorhandener ändert.*

Folgendes Diagramm gibt den Zusammenhang wieder zwischen Normalordnung und Übersetzung:

Wenn t nach $\tau(t)$ übersetzt wird, und man einen Normalordnungsreduktionsschritt von t nach t' machen kann, dann kann man $\tau(t)$ in KFP-Normalordnung auf $\tau(t')$ reduzieren, wie im Diagramm angegeben.

$$\begin{tikzcd} t \arrow[r] \arrow[d, "no"] & \tau t \arrow[d, "no,*"] \\ t' \arrow[r] & \tau t' \end{tikzcd}$$

Insgesamt haben wir damit eine Übersetzung τ von KFP+-Superkombinatoren nach KFP, so dass die Terminierung und die kontextuelle Approximation erhalten bleibt, D.h. Wir können sagen, dass KFP und KFP+ äquivalent sind:

Satz 7.17.

- $t\Downarrow_{KFP+} \Leftrightarrow \tau(t)\Downarrow$
- $s \leq_{c,KFP+} t \Leftrightarrow \tau(s) \leq_c \tau(t)$

Begründung des Satzes. Dass die Terminierung erhalten bleibt, folgt aus den Überlegungen zur Übertragung der Reduktion.
Die Übersetzung τ ist die Inverse zu σ: Es gilt $\tau(\sigma(t)) \sim_c t$.
Den zweiten Teil zeigen wir für geschlossene Ausdrücke s, t:

- $s \leq_{c,KFP+} t \Rightarrow \tau(s) \leq_c \tau(t)$:
  Sei C ein Kontext in KFP, so dass $C[\tau(s)], C[\tau(t)]$ geschlossen sind. Die Definition der kontextuellen Gleichheit in KFP+ zeigt, und die Übersetzung σ, angewendet auf den Kontext C, zeigen, dass man ein Programm $P'[]$ definieren kann mit $main = C'[]$, so dass $main_{P'[s]}\Downarrow \Rightarrow main_{P'[t]}\Downarrow$.
- $\tau(s) \leq_c \tau(t) \Rightarrow s \leq_{c,KFP+} t$:
  Sei $P'[]$ ein Oberprogramm-Kontext des aktuellen Programms. Die Übersetzung σ erlaubt jetzt, einen Kontext C in KFP zu definieren, der die Auswertung von $main_{P'[s]}$ und $main_{P'[s]}$ widerspiegelt.

Für offene Ausdrücke benötigt man etwas mehr Aufwand, den wir hier nur skizzieren: Beachte, dass In KFP gilt:

$s \leq_c t$ gdw. $\forall C$: wenn $C[s], C[t]$ geschlossen, dann $C[s]\Downarrow \Rightarrow C[t]\Downarrow$.
$s \leq_c t$ gdw. $\lambda x_1, \ldots, x_n.s \leq_c \lambda x_1, \ldots, x_n.t$.

- $s \leq_{c,KFP+} t \Rightarrow \tau(s) \leq_c \tau(t)$:
  Sei C ein Kontext in KFP, so dass $C[\tau(s)], C[\tau(t)]$ geschlossen sind, und der im Inneren zunächst die freien Variablen in ein Lambda einbindet. D.h. der innere Kontext ist $\lambda x_1, \ldots, x_n.[])$. Den Kontext kann man so wählen, dass er nicht mehr lambdageliftet werden muss. Die Definition der kontextuellen Gleichheit in KFP+ zeigt, und die Übersetzung σ, angewendet auf den Kontext C, zeigen, dass man ein Programm $P'[]$ definieren kann mit $main = C'[]$, so dass $main_{P'[s]}\Downarrow \Rightarrow main_{P'[t]}\Downarrow$.

– $\tau(s) \leq_c \tau(t) \Rightarrow s \leq_{c,KFP+} t$:
Sei $P'[]$ ein Oberprogramm-Kontext des aktuellen Programms. Die Übersetzung σ erlaubt jetzt, einen Kontext C in KFP zu definieren, der die Auswertung von $main_{P'[s]}$ und $main_{P'[s]}$ widerspiegelt.

□

Satz 7.18. *KFP+ und KFP sind äquivalent bzgl. der Übersetzungen τ und σ.*

Definition 7.19. *Seien L, M zwei Programmiersprachen mit jeweils einer Reduktion und zugehörigen kontextuellen Ordnung. Von einer* Implementierung *$\rho : L \to M$ spricht man bei einer Übersetzung mit folgenden Eigenschaften:*

- *ρ soll natürlich effektiv sein.*
- $\forall t : t\Downarrow \Leftrightarrow \rho(t)\Downarrow_M$
- $\forall t : t \to t' \Rightarrow \rho(t) \to \rho(t')$
- $\forall s, t : \rho(s) \leq_{c,M} \rho(t) \Rightarrow s \leq_c t$. *Die Rückrichtung könnte falsch werden, da es in der Zielsprache mehr Ausdrücke geben könnte.*
- *Die kontextuelle Approximation soll erhalten bleiben (in der Menge der Bilder).*

Bemerkung 7.20. Bei Äquivalenzen muß man manchmal subtile Unterschiede beachten. Z.B. ist die Behauptung, dass eine Übersetzung das Terminierungsverhalten nicht ändert, etwas anderes als die Behauptung, dass die kontextuelle Äquivalenz erhalten bleibt, da sich möglicherweise die Kontexte nicht ganz entsprechen.

8 Church-Zahlen: KFP ist nicht minimal

Das Konstrukt `case` und Konstruktoren sind nicht unbedingt notwendig, wobei hier gemeint ist, dass man `case` und Konstruktoren in KFP ohne Konstruktoren implementieren kann. Die kontextuelle Äquivalenz geht dabei möglicherweise verloren.
Als erstes Beispiel können wir Funktionen definieren, die sich wie die Booleschen Funktionen verhalten: Danach die Zahlen, wie A. Church sie definiert hat Church-Zahlen).

1. man definiert Boolesche Funktionen:

$$\begin{array}{ll} TRUE\ x\ y & = x \\ FALSE\ x\ y & = y \\ IF\ x\ y\ z & = (x\ y\ z) \\ AND\ x\ y & = x\ (y\ TRUE\ FALSE)\ FALSE \\ OR\ x\ y & = x\ TRUE\ y \\ NOT\ x & = x\ FALSE\ TRUE \end{array}$$

2. Paare und auch Tupel:

$$\begin{aligned}
PAIR\ x\ y\ z &= z\ x\ y \\
FST\ x &= x\ TRUE \\
SND\ x &= x\ FALSE
\end{aligned}$$

3. Zahlen (nach A. Church.) Die Darstellung der Zahlen ist: $zahl\ s\ z = s\ (s\ \ldots\ (s\ z)\ \ldots)$

$$\begin{aligned}
ZERO\ x\ y &= y \\
SUCC\ n\ s\ z &= s\ (n\ s\ z) \\
PLUS\ m\ n &= m\ SUCC\ n \\
TIMES\ m\ n &= m\ (PLUS\ n)\ ZERO \\
EXPT\ b\ e &= e\ (TIMES\ b)(SUCC\ ZERO) \\
PREDR\ p &= PAIR\ (SUCC\ (FST\ p))(FST\ p) \\
PRED\ n &= SND\ (n\ PREDR\ (PAIR\ ZERO\ ZERO)) \\
MINUS\ m\ n &= n\ PRED\ m \\
ZERO?\ n &= n\ (TRUE\ FALSE)\ TRUE \\
EQUAL?\ m\ n &= (AND\ (ZERO?\ (MINUS\ m\ n)) \\
&\quad (ZERO?\ (MINUS\ n\ m)))
\end{aligned}$$

$(PAIR\ x\ y)$ soll die Paarbildung nachbilden. Beachte, dass ZERO und FALSE diesselbe Funktion sind.

Übungsaufgabe 8.1. Berechne mit obigen Definitionen $1 = 1$ und $1 = 2$:

Man kann nachprüfen, daß folgendes gilt:

- $PRED\ (SUCC\ n) = n$, wenn n von der Form $(SUCC^k\ ZERO)$ ist.
- $PRED\ ZERO = ZERO$, d.h. $PRED$ berechnet für positive Zahlen n die nächstkleinere Zahl und für die $Null$ ist das Resultat jeweils $Null$.

Als Beispiel berechnen wir: $(PRED\ ZERO)$ und kürzen ab: $TRUE = T, FALSE = F, ZERO = Z$

$$\begin{aligned}
(PRED\ Z) &\to SND\ (Z\ PREDR\ (PAIR\ Z\ Z)) \\
&\to (Z\ PREDR\ (PAIR\ Z\ Z))\ F \\
&\to (PAIR\ Z\ Z)\ F \\
&\to F\ Z\ Z \\
&= Z
\end{aligned}$$

$$\begin{aligned}
ZERO?\ ZERO &\to ZERO\ (T\ F\ )\ T \\
&\to T
\end{aligned}$$

$$\begin{aligned}
ZERO?\ (SUCC\ ZERO) &\to SUCC\ Z\ (T\ F)\ T \\
&\to T\ F\ (Z\ (T\ F)\ T) \\
&\to F
\end{aligned}$$

Will man negative Zahlen codieren, so kann man dies durch Zahlen mit Vorzeichen in der folgenden Form durchführen: $+n$ als $(PAIR\ TRUE\ n)$ $-n$ als $(PAIR\ FALSE\ n)$

Übungsaufgabe 8.2. Definition der Rechenoperationen auf diesen ganzen Zahlen.

Bemerkung 8.3. Was bleibt (nicht) erhalten bei dieser Übersetzung?
Ungetypte Ausdrücke werden teilweise in terminierende übersetzt, da das case nicht exakt übersetzbar ist. Die kontextuelle Ordnung bleibt ebenfalls nicht erhalten, da $Y\ K$ nach der Übersetzung zum größten Element wird.
Was erhalten bleibt, ist die Eigenschaft:
Wenn $t \xrightarrow{no,*} t'$ wobei t' in WHNF, dann auch $\tau(t) \xrightarrow{no,*} \tau(t')$, wobei $\tau(t')$ in WHNF. D.h. nur erfolgreiche Berechnungen werden korrekt übersetzt.
D.h. diese Übersetzung ist keine volle Übersetzung der gesamten Sprache KFP+, sondern eine, die Berechnungen simulieren kann.
Betrachtet man nur die Übersetzung der getypten Ausdrücke, so könnte sich die Situation etwas verbessern: unklar.

8.1 SKI-Kombinatoren

Es gibt eine Übersetzung des Lambda-Kalküls in eine einfache Superkombinatorsprache, bei der ebenfalls die Äquivalenz erhalten bleibt: die sogenannte SKI-Übersetzung. Die Sprache KFP (KFP+) muss man erst vorbehandeln, d.h. in den Lambdakalkül übersetzen, um danach diese Übersetzung verwenden zu können.
Dazu benötigen wird drei Kombinatoren S, K, und I:

$$\begin{aligned} S\ x\ y\ z &= (x\ z)\ (x\ z) \\ K\ x\ y\ &= x \\ I\ x\ &= x \end{aligned}$$

Diese Definitionen können offenbar in KFP+, also auch in KFP durchgeführt werden. Die Reduktion ist wie in KFP+. Die Übersetzung von reinen Lambda-Termen benötigt die jeweils abstrahierte Variable und eine erweiterte Zwischen-Darstellung von Ausdrücken.
$[x]E$ bedeutet: die Variable x soll aus E abstrahiert werden und hat Wirkung analog zu $(\lambda x.E)$. Dies kann auch verschachtelt sein.
λ-Ausdrücke werden so übersetzt:

$$\lambda x.E \to ([x]\ E)$$

Danach wirken die folgenden drei Übersetzungsregeln:

1) $[x]\ (a\ b) \to S\ ([x]\ a)\ ([x]\ b)$
2) $[x]\ x \quad \to I$
3) $[x]\ y \quad \to K\ y$ (y Variable oder Konstante, die von x verschieden ist. Die Konstanten hier nur:S, K, I)

Beispiel 8.4. Wir übersetzen
$TRUE\ x\ y = \lambda x.(\lambda y.x))$ in SKI:
$[x]\ ([y]\ x) \to\ [x]\ (K\ x) \to\ S\ ([x]\ K)([x]\ x) \to\ S\ (K\ K)\ I$
Die Reduktion von $(TRUE\ a)$ mittels SKI ergibt:
$S\ (K\ K)\ I\ a \to (K\ K\ a)\ (I\ a) \to K\ a$

Man kann jetzt zeigen, daß die Wirkung des Originalausdrucks und des übersetzten Ausdrucks identisch ist.
Die β-Reduktion wird durch die SKI-Kombinatoren implementiert:
$(\lambda x.E)\ t \to E[x/t]$. Die übersetzte Version leistet das gleiche: $([x]E)\ t'$ hat drei Möglichkeiten:

1. E ist Applikation. Dann wurde dies mittels S übersetzt:
   $([x]\ (a\ b))\ t' \to\ S\ ([x]\ a)([x]\ b)\ t' \to ([x]\ a)\ t')(([x]\ b)\ t')$
2. E ist die Variable x:
   $([x]\ x)\ t' \to\ I\ t' \to t'$
3. E ist Konstante oder eine Variable $\neq x$ $([x]\ y)\ t' \to\ K\ y\ t' \to y$

Man kann zeigen, dass α-äquivalente (bis auf Umbenennung gleiche) Lambda-Ausdrücke in den gleichen SKI-Kombinatorausdruck übersetzt werden, und daß Namenskonventionen bei der durch SKI simulierten β-Reduktion erhalten bleiben. Kontextuelle Gleichheit ist übertragbar.
Was auch noch gilt, ist die Eigenschaft, daß irreduzible Lambda-Ausdrücke in irreduzible SKI-Ausdrücke übersetzt werden.
Die Anzahl der Reduktionsschritte in beiden Kalkülen ist verschieden, da in SKI auch die Ersetzung im Term als mehrere Reduktionsschritte gezählt werden.

9 Approximationen in KFP und Induktion

Auf der Ordnung $\leq_c$ in KFP kann man aufbauen, um mittels Approximation kontextuelle Gleichheit von interessanten Ausdrücken in KFP zu zeigen. Da wir bereits argumentiert haben, dass die kontextuelle Gleichheit in KFP+ genauso verwendet werden kann wie die kontextuelle Gleichheit in KFP, werden wir die Unterschiede ignorieren.

Beispiel 9.1. Wir hatten bereits in einem Beispiel vermutet, dass die Kette $\bot \leq_c$ `(S bot)` $\leq_c$ `(S (S bot))` $\leq_c \ldots$ gegen den unendlich verschachtelten Term `S (S (S (... )))` konvergiert. Insbesondere, dass alle Ausdrücke, die bei Auswertung einen solchen Term produzieren, kontextuell gleich sind.

Definition 9.2. *Sei $s_1 \leq_c s_2 \leq_c \ldots$ eine aufsteigende Kette. Der Ausdruck s ist eine* kleinste obere Schranke (lub) *dieser Kette, gdw.*
$\forall i : s_i \leq_c s$ und für alle r gilt: $(\forall i : s_i \leq_c r) \Rightarrow s \leq_c r$.

Dieser Begriff des lub reicht für unsere Zwecke nicht aus. Man muß noch etwas mehr definieren:

Definition 9.3. *Sei $s_1 \leq_c s_2 \leq_c \ldots$ eine aufsteigende Kette. Der Ausdruck s ist eine* kontextuelle kleinste obere Schranke (club) *dieser Kette, gdw $\forall C : C[s] = lub_i(C[s_i])$. Wir notieren das als $s = club_i(s_i)$.*

Die Begriffe lub und club sind nur eindeutig bis auf $\sim_c$ definiert.
Es gibt folgendes Kriterium für einen club einer aufsteigenden Kette:

Lemma 9.4. *Sei $s_1 \leq_c s_2 \leq_c \ldots$ eine aufsteigende Kette, sei s ein Ausdruck. Es gelte:*

1. *Für alle $i : s_i \leq_c s$*
2. *Für alle Kontexte C gilt: $C[s]\Downarrow \Rightarrow \exists i : C[s_i]\Downarrow$*

Dann gilt $s = club_i(s_i)$.

Beweis. Sei D ein Kontext. Wir wollen zeigen $D[s] = lub_i(D[s_i])$. Sei dazu C ein weiterer Kontext und r ein Ausdruck mit $\forall i : D[s_i] \leq_c r$. Nach dem Kriterium gilt: Wenn $CD[s]\Downarrow$, dann existiert ein j mit $CD[s_j]\Downarrow$, also wegen $D[s_i] \leq_c r$ gilt dann auch $C[r]\Downarrow$. Es gilt also: $CD[s]\Downarrow \Rightarrow C[r]\Downarrow$. Da das für alle Kontexte C gilt, haben wir $D[s] \leq_c r$ gezeigt. □

Übrigens ist es mW offen, ob jeder lub auch ein club ist.

Bemerkung 9.5. Nicht jede aufsteigende Kette hat einen club in der Menge der Ausdrücke. Z.B. kann man für jeden unendlichen String $a_1 a_2 a_3 \ldots$ über $\{0, 1\}$ eine aufsteigende Kette bilden: $bot \leq_c a_1 : bot \leq_c a_1 : a_2 : bot \leq_c a_1 : a_2 : a_3 : bot \leq_c$. Man kann mit den bisher vorgestellten Mitteln nachweisen, dass der club im Falle der Existenz eindeutig ist, gerade der "unendlichen Liste" $a_1 : a_2 : \ldots$ entspricht und für verschiedene Strings verschiedene clubs produziert. Allerdings ist die Menge der möglichen Ausdrücke abzählbar, während die Menge der unendlichen Strings überabzählbar ist. Damit muß es aufsteigende Ketten geben, die keinen club besitzen.
Man kann auch konkrete aufsteigende Ketten angeben ohne club indem man nichtberechenbare Funktionen verwendet.

Definition 9.6. *Ein Abbildung ϕ, die Ausdrücke auf Ausdrücke abbildet heißt* stetig, *gdw*

1. $s \leq_c t \Rightarrow \phi(s) \leq_c \phi(t)$
2. *Sei s_i aufsteigende Kette mit $club_i s_i = s$. Dann ist $club_i(\phi(s_i)) = \phi(s)$*

Aussage 9.7. *Jeder KFP-Ausdruck ist eine stetige Abbildung. D.h. die Abbildung ϕ_t mit:*

$$\phi_t\ s := t\ s$$

ist stetig.

Beweis. Folgt direkt aus den Definitionen, da $(t\ .)$ ein Kontext ist. □

Definition 9.8. *Zwei Ausdrücke s, t sind* gleich bis auf Rechtstiefe n, *wenn jede Position, an der s, t verschiedene Markierung haben, einen Teilpfad der Form $\underbrace{2 \ldots 2}_{m}$ mit $m \geq n$ hat. D.h. der Pfad von der Wurzel zur Position hat ein Teilstück, das mindestens n mal nach rechts geht. Diese Definition machen wir modulo α-Gleichheit.*

Beispiel 9.9. Die Ausdrücke $\underbrace{f(\ldots(f}_{8}\ a)\ldots)$ und $\underbrace{f(\ldots(f}_{8}\ b)\ldots)$ unterscheiden sich an der Markierung a, b, die eine Rechtstiefe von 8 hat. Somit sind die beiden Terme gleich bis auf Rechtstiefe 8.

Lemma 9.10.

- *Wenn zwei Ausdrücke s_1, s_2 gleich sind bis auf Rechts-tiefe n, und $s_1 \rightarrow_{no} s'_1, s_2 \rightarrow_{no} s'_2$, dann sind s'_1, s'_2 gleich bis auf Rechtstiefe $n-1$.*
- *Wenn zwei Ausdrücke s_1, s_2 gleich sind bis auf Rechts-tiefe n, und $s_1 \rightarrow_{no,m} s'_1, s_2 \rightarrow_{no,m} s'_2$, dann sind s'_1, s'_2 gleich bis auf Rechtstiefe $n-m$.*
- *Wenn zwei Ausdrücke s_1, s_2 gleich sind bis auf Rechts-tiefe $n \geq 1$, und s_1 ist eine WHNF, dann ist auch s_2 eine WHNF.*

Beweis. Die jeweiligen Normalordnungsredexe von s_1, s_2 sind ebenfalls gleich bis auf Rechtstiefe n.
Eine Beta-Reduktion verändert folgendes: $\lambda x.t_1\ r_1 \rightarrow t_1[r_1/x]$. Die Ausdrücke r_1, r_2 sind gleich bis auf Rechts-tiefe $n-1$. Die Ausdrücke $t_1[r_1/x]$ und $t_2[r_2/x]$ sind gleich bis auf Rechtstiefe $n-1$; Jeder Pfad, der in Rechtstiefe $\leq n-1$; verschiedene Markierungen findet, kann nicht ganz in t_1, t_2 sein, sondern geht durch eine ehemalige Position eines x, und folgt dann (einer Kopie von) r_1 bzw r_2.
Für eine Case-Reduktion gilt das analog, da die Änderungen ebenfalls Ersetzungen sind. □

Satz 9.11. *Sei t ein (evtl. offener) Ausdruck, und sei $t_0 := \bot, t_1 = (t\ bot), t_i = t(t_{i-1})$. Dann ist $Y\ t \sim_c club_i(t_i)$. Außerdem ist $Y\ t$ kleinster Fixpunkt von t.*

Beweis.

- t_i ist eine aufsteigende Kette: Mit Induktion nach i kann man zeigen, dass t_i aufsteigend ist: $\bot \leq_c t_1$. Dann mit Induktion: aus $t_{i-1} \leq_c t_i$ folgt mit der Monotonie: $t\ t_{i-1} \leq_c t\ t_i$, d.h. $t_i \leq_c t_{i+1}$.
- $\forall i : t_i \leq_c Y\ t$. Es gilt $t_0 \leq_c Y\ t$.
  Induktionsschritt: Sei $t_n \leq_c Y\ t$. Dann gilt auch $t\ t_n \leq_c t(Y\ t)$, da $\leq_c$ eine Präkongruenz ist. Das bedeutet aber gerade $t_{n+1} \leq_c Y\ t$.
- $Y\ t \leq_c club_i(t_i)$. Wir weißen das Kriterium in Lemma 9.4 nach.
  Sei C ein Kontext, so dass $C[Y\ t]\Downarrow$ und sei i die Anzahl der notwendigen Normalordnungsreduktionen. Dann gilt $C[t_i]\Downarrow$. Dazu ersetzen wir $C[Y\ t]$ durch $C[t^{i+1}(Y\ t)]$. Dieser Term hat nach Aussage 5.5 nicht mehr als i Normalordnungsreduktionen zur WHNF. Er ist gleich zu $C[t_{i+1}]$ bis auf Rechtstiefe $i+1$. Nach Lemma 9.10 ist die WHNF von $C[t^{i+1}(Y\ t)]$ gleich bis auf Rechtstiefe 1 zu dem Ausdruck, der nach i Normalordnungsreduktionen aus $C[t_{i+1}]$ entsteht. Das bedeutet, $C[t_{i+1}]$ ist ebenfalls eine WHNF.
  Jetzt können wir schließen, dass $Y\ t \sim_c club_i(t_i)$.

– $Y\ t$ ist kleinster Fixpunkt von t: Sei f ein Fixpunkt von t. Dann gilt $t\ f \sim_c f$. Mit Induktion kann man leicht zeigen, dass $t_i \leq_c f$:
  Es ist $bot \leq_c f$; und wenn $t_i \leq_c f$, dann auch $t\ t_i \leq_c t\ f$, d.h. $t_{i+1} \leq_c f$. Aus der Eigenschaft der lubs folgt jetzt $Y\ t \leq_c f$. D.h. $Y\ t$ ist der kleinste Fixpunkt von t (bzgl. $\leq_c$).

$\square$

Bemerkung Die Ausdrücke t_i entsprechen der rekursiv definierten Funktion t, wobei das erste Argument der Name der Funktion ist, wobei man die Rekursion nach i Schritten abbricht, und dann bot evaluiert. D.h. eine rekursive Funktion wird approximiert durch Funktionen, die jeweils immer mehr rekursive Aufrufe machen dürfen.

Beispiel 9.12. Sei `S` ein einstelliger Konstruktor. Jetzt können wir zeigen, dass alle Ausdrücke t, die einen unendlichen Ausdruck `(S (S (S ...)))`, mittels Reduktion erzeugen, kontextuell äquivalent sind. Genauer: Ausdrücke t mit der Eigenschaft: $\forall n.\exists t' : \underbrace{S(\ldots(S}_{n}\ t')\ldots)$.

Das Theorem zeigt, dass $(\mathtt{Y}\ \mathtt{S}) \sim_c club(s_i)$, wobei $s_i = \underbrace{(\mathtt{S}(\ldots(\mathtt{S}}_{i}\bot)))$. Damit erhält man auch, dass $(\mathtt{Y}\ \mathtt{S}) \leq_c t$, da $s_i \leq_c t$ für jede Approximation s_i gilt. Es fehlt noch der Nachweis $t \leq_c (\mathtt{Y}\ \mathtt{S})$. Den kann man analog zum Beweis von Satz 9.11 ausführen, indem man mit der Rechtstiefe argumentiert.

9.1 Nachweis von Gleichungen: Induktion

Will man eine Eigenschaft einer Definition oder eines Ausdrucks nachweisen, so kann man dies mit einer speziellen Form der Induktion machen:
Fixpunktinduktion.

> Sei P ein (totales oder partielles) Prädikat. Will man $P(s)$ nachweisen, bzw. $\forall x_1, \ldots, x_n.P(s)$, so kann man in vielen interessanten Fällen diesen Nachweis führen, indem man eine Folge s_i angibt mit $club_i s_i = s$, und $P(s_i)$ für alle i nachweist; d.h. $P(s_i) =$ `True`. Wenn P stetig als Funktion von Ausdrücken nach `Bool` ist, dann kann man daraus schließen, dass $P(s)$ gilt. Stetige Prädikate nennt man auch *zulässige Prädikate.*

Meist sind diese Prädikate Gleichungen zwischen Ausdrücken, wobei die Semantik der Gleichungen gerade $\sim_c$ ist. Man sieht leicht ein, daß $\sim_c$ als Funktion von zwei Argumenten stetig ist, denn wenn s_i aufsteigende Kette, t_i aufsteigende Kette, und $s_i \sim_c t_i$ für alle i, dann ist auch $s \sim_c t$, wenn $s = club_i s_i$ und $t = club_i t_i$. Damit haben wir eine Klasse von zulässigen Prädikaten. Man kann weitere zulässige Prädikate bilden durch Zusammensetzen von anderen zulässigen, indem man die logischen Operatoren $\wedge, \vee$, verwendet. Allerdings ist *not* nicht erlaubt.

Der Nachweis, dass $P(s_i)$ für alle i gilt, wird oft in Form einer Induktion durchgeführt. Oft kann man die Form der möglichen Argumente rekursiv beschreiben, wobei man alle WHNFs aufzählt.
Eine Standardform der Induktion ist der Nachweis von Eigenschaften für Funktionen, die Listenargumente haben. Induktionsschema zum Nachweis von $P(.)$ für Listen:

- Zeige $P(\perp)$.
- Zeige $P(Nil)$.
- Zeige: $P(xs) \Rightarrow P(s : xs)$.

Dann kann man schließen: $P(s)$ gilt für alle Listen s.

Definition 9.13. *Sei eine Funktion zur Berechnung der "Partial-Liste" einer Liste definiert als*

```
partlist 0 xs = bot
partlist (n+1) [] = []
partlist (n+1) (x:xs) = x: partlist n xs
```

Beispiel 9.14. Die List $l =$ `[1..]` hat als Partiallisten `\bot`, `1:bot`, `1:2:bot`.

Lemma 9.15. *Sei eine (unendliche) Liste $l \neq \perp$ gegeben und sei $l_i :=$* `partlist i l`. *Dann ist l_i aufsteigend bzgl. $\leq_c$ und $club_i(l_i) = l$.*

Beweis. Wir nehmen an, dass sich die Liste l beliebig tief in Richtung "Tail" auswerten läßt. Anderenfalls wäre der n-te Tail der Liste = bot.
Es ist klar, dass $l_i \leq_c l$. Um $l = club_i l_i$ zu zeigen, müssen wir zeigen, dass für jeden Kontext $C[]$: $C[l]\Downarrow \Rightarrow \exists i : C[l_i]\Downarrow$.
Wir verwenden dazu die Rechtstiefe. Sei C ein Kontext, so dass $C[l]\Downarrow$. Sei i die Anzahl der Normalordnungsschritte, um $C[l]$ zu einer WHNF zu reduzieren. Wir können annehmen, dass l so ausgewertet ist, dass l die Form $a_1 : a_2 : \ldots a_{i+1} : a_{i+2}$ hat, wobei a_{i+2} auch unausgewertet sein kann. Sei l_i, definiert wie oben, bereits so weit ausgewertet, dass $l_i = a_1 : a_2 : \ldots a_{i+1} : \perp$. Dann sind l_i und l gleich bis auf Rechtstiefe $i+1$. Lemma 9.10 zeigt dann, dass $C[l_i]$ ebenfalls nach i Normalordnungsschritten zu einer WHNF wird. D.h. $C[l_i]\Downarrow$.
Damit ist das Kriterium von Lemma 9.4 erfüllt und wir können schließen, dass $l = club_i(l_i)$ ist.

□

Beispiel 9.16. Eine Ungleichung ist i.a. nicht zulässig:
Die Ungleichheit $\neq$ ist nicht stetig: Sei $l =$ `repeat 1` und $l_i =$ `partlist i l`. Dann gilt $l_i \not\sim_c l$ für alle i, aber $club_i(l_i) = l$. Wäre $\neq$ stetig, dann ergäbe sich $l \not\sim_c l$, was Unsinn ist.
Das Induktionsschema für unendliche Listen funktioniert für das Prädikat $P(x) \equiv x \neq l$. Offenbar gilt $bot \not\sim_c l$ und $[] \not\sim_c l$. Nimmt man $xs\ //sim_c l$ als Induktionshypothese und will man $x : xs \not\sim_c l$ zeigen, so gibt es zwei Fälle. Wenn $x \not\sim_c 1$, dann ist das offenbar richtig. Wenn $x \sim_c 1$, dann erfordert $1 : xs \not\sim_c l$ wegen $l = 1 : l$ gerade $xs \not\sim_c l$. Das war die Induktionsbehauptung.

Begründung des Induktionsschemas Stetigkeit und *club* werden benutzt. Da alle Listen sich als *club* von partiellen Listen darstellen lassen, kann man für jede Liste l eine aufsteigende Folge l_i mittels `partlist` definieren, die diese Liste als *club* hat.

- Ist die Liste endlich, dann zeigt das Induktionsschema die Aussage direkt mit normaler Induktion
- Ist die Liste unendlich, dann zeigt das Induktionsschema die Aussage $P(l)$ für alle partiellen Listen l_i = `partlist l` mit normaler Induktion. Da das Prädikat zulässig ist, darf man wegen $l = club_i(l_i)$ schließen, dass $P(l)$ gilt.

Beispiel 9.17. Zeige `append [] xs` $\sim_c$ `xs`. Die Verwendete Definition von `append` ist.

```
(append)                :: [a] -> [a] -> [a]
append []       ys      = ys
append (x:xs)   ys      = x : (append xs ys)
```

Da `append` mit Fallunterscheidung über das erste Argument definiert ist, kann man das sofort zu `xs` reduzieren.

Beispiel 9.18. Zeige `append xs []` $\sim_c$ `xs`. Da `append` mit Fallunterscheidung über das erste Argument definiert ist, ist das nicht direkt mit Reduktion zu zeigen. Wir benutzen das Induktionsschema für das erste Argument.

- `append bot []` $\sim_c$ `bot`, da die Auswertung von `bot` durch die case-Reduktion nicht terminiert.
- `append [] []` $\sim_c$ `[]` mittels Reduktion
- `append (x:xs) []` reduziert zu `x: (append xs [])`. Nach Induktionshypothese ist `(append xs [])` $\sim_c$ `xs`. Da $\sim_c$ sich über Konstruktoren ziehen läßt, erhalten wir `append (x:xs) []` $\sim_c$ `x:xs`.

Insgesamt erhalten wir jetzt mit dem Induktionsschema die Behauptung.

Hiermit können wir jetzt weitere Gleichheiten zeigen:

Beispiel 9.19. `append` ist assoziativ. Es gilt: `(append (append bot xs) ys)`= *bot* = `(append bot append(xs ys))`
Es gilt: `(append (append [] xs) ys)`= `(append xs ys)` = `(append [] append(xs ys))`

Es gilt: `(append (append (z:zs) xs) ys)`= `(append (z : append zs xs) ys)`
= `z : (append (append zs xs) ys)`
= `z : (append zs (append xs ys))` (mit Induktionshypothese)
= `(append (z:zs) (append xs ys))`.
Danach können wir Induktion verwenden.

Definition 9.20. *Ein Induktionsschema zum Nachweis von Eigenschaften P endlicher Listen, wobei P nicht stetig zu sein braucht.*

– *Zeige* $P([])$
– *Zeige* $P(xs) \Rightarrow P(x : xs)$.

Dann kann man $P(xs)$ *für alle endlichen Listen schließen.*

Definition 9.21. *Ein weiteres Induktionsschema zum Nachweis von (beliebigen) Eigenschaften* P *endlicher Listen kann man auf der Länge der Listen aufbauen.*

– *Zeige:* $P([])$

– *Zeige:* $\begin{array}{l} P(xs) \text{ für alle Listen der Länge } n \\ \Rightarrow P(ys) \text{ für alle Listen der Länge } n+1 \end{array}$

Dann kann man $P(xs)$ *für alle endlichen Listen schließen.*

Beispiel 9.22. Für alle endlichen Listen gilt:

– `length(xs ++ ys)` = `(length xs) + (length ys)`
Das zeigt man mit Induktion nach der Länge der Listen:
Wenn $xs = []$, dann gilt die Behauptung.
`length((x:xs) ++ ys)` reduziert zu `length(x:(xs ++ ys)`; dies reduziert zu `1+ length(xs ++ ys)`. Mit Induktion ist das gleich `1+ length(xs) + length ys)`. Da `length((x:xs)` zu `1 + length xs` reduziert und + assoziativ ist, erhält man die Behauptung.
– `length (reverse xs)`= `length xs`
Für Listen der Länge 0 gilt die Behauptung.
Für Listen der Länge > 0 erhält man:
`length (reverse (x:xs))`→ `length (reverse xs) ++ [x])`
Mit obiger Gleichung erhält man: = `length (reverse xs) + 1)`
Induktion nach der Länge ergibt: = `(length xs) + 1)` = `(length x:xs)`

– `reverse (reverse xs)` = `xs`.
Das ist komplizierter: Man benötigt als Zwischenbehauptungen: `++` ist assoziativ und `reverse(xs ++ ys)` = `(reverse ys) ++ (reverse xs)`. Die sind jeweils mit Induktion zu zeigen.
Danach kann man für nichtleere Listen schließen:
`reverse (reverse (x:xs)` = `reverse ((reverse xs) ++ [x])`
= `reverse [x] ++ (reverse(reverse xs))` = `[x] ++ xs` = `x:xs`.

Aber für unendliche Listen ist die letzte Aussage falsch, denn `reverse (repeat 1)` = `bot` und `reverse bot` = `bot`.

Bemerkung 9.23. `reverse (reverse xs) = xs` ist ein Beispiel für ein stetiges Prädikat, das sowohl für alle endlichen Listen als auch für $xs = bot$ gilt, aber nicht für alle Listen. Der Beweis, dass `reverse (reverse xs) = xs` für alle endlichen lIsten gilt wurde nicht nach dem Schema für alle Listen, sondern nach der Länge der Listen geführt.

Induktionsschema für alle Ausdrücke: Eine Erweiterung der Induktion kann zum Nachweis von stetigen Eigenschaften $P(\cdot)$ benutzt werden, wobei alle Ausdrücke als Argumente zulässig sind. Insbesondere kann dieses Schema zum Nachweis von Gleichungen benutzt werden.

- Zeige $P(\bot)$.
- Zeige $P(\lambda x.r)$.
- Zeige für alle Konstruktoren c: $P(x_1) \wedge \ldots \wedge P(x_n) \Rightarrow P(c\ x_1 \ldots x_n))$ wobei $n = ar(c)$.

Dann kann man schließen: $P(s)$ gilt für alle Ausdrücke s.

Bemerkung 9.24. Das Schema scheint nicht die Verallgemeinerung des Induktionsschemas für alle Listen zu sein. Aber es ist doch konsistent dazu, denn der Induktionsschritt $P(x) \wedge P(xs) \Rightarrow P(x : xs)$ paßt trotzdem, denn im Falle der Listen zeigt man den Schritt für alle x. Im Falle der Betrachtung aller Ausdrücke zeigt man ja gerade $P(x)$ für alle Ausdrücke x.

Beispiel 9.25. Am Beispiel der Assoziativität von `append` kann man zeigen, dass die Assoziativitätsgleichung für alle Ausdrücke gilt.

- Für $s = (\lambda x.r)$ erhält man auf beiden Seiten $\sim_c bot$, da ein Typfehler auftritt.
- Für $s = c\ x_1 \ldots x_n$ erhält man ebenfalls auf beiden Seiten $\sim_c bot$, da ein Typfehler auftritt.
- die anderen Fälle wurden bereits gezeigt.

Bemerkung 9.26. Diese Gleichheiten kann man verwenden, um Programme in korrekter Weise zu verändern. Z.B. darf man die Assoziativität von append verwenden, um beliebige append-Ausdrücke umzuklammern. Dies wäre falsch, wenn die Assoziativität nur für endliche Listen gelten würde.
Wenn klar ist, dass an einer Stelle im Programm nur endliche Listen verwendet werden können, darf man dort auch die Gleichheiten verwenden, die für endliche Listen gelten. Z.B. wenn nach der Abfrage `(liste-endlich l)` die Variable `l`verwendet wird, dann kann man das wie eine endliche Liste behandeln, denn im Falle einer unendlichen Liste ergibt sich `bot` bereits bei der Abfrage.

Übungsaufgabe 9.27. Welche der folgenden Gleichungen gelten für endliche, welche für unendliche Listen?

- `xs ++ []` = `xs`?
- `concat (xs ++ ys)` = `(concat xs) ++ (concat ys)`?
- `(take n xs) ++ (drop n xs)` = `xs`?
- Für Listen xs, ys gleicher Länge: `unzip (zip xs ys)` = `(xs,ys)`? mit den zwei Definitionsvarianten

```
unzip_1  = foldr (\(a,b) paar ->
                   (a:(fst paar), b:(snd paar))) ([], [])
unzip_2  = foldr (\(a,b) (as,bs) -> (a:as, b:bs)) ([], [])
```

- `map f (map g xs)` = `map (f.g) xs` (Verschmelzungsgesetz für `map`)

9.2 Gleichungen für (un-)endliche Listen: `map, concat, foldr, foldl`

Das Buch von Richard Bird, 98 enthält einige Gleichungen und Gesetze zum Zusammenhang zwischen `foldr, foldl, map, concat` und weiteren Funktionen. Diese Gleichungen können für verschiedene Nachweise der Korrektheit und für Optimierungen verwendet werden. Im folgenden einige dieser Gleichungen:

Erstes Dualitätsgesetz für FOLD Sei f die binäre Verknüpfung eines Monoids mit dem Einselement e. D.h. als Funktion sei f eine zweistellige Funktion, e ein Ausdruck, so dass modulo $\sim_c$ die Monoidgleichungen gelten: Assoziativität und Einselement. Beachte, dass in Monoiden *bot* als Element vorkommen kann. I.a. wird bot ein Nullelement des Monoids sein.
Dann gilt für alle endliche Listen xs mit Elementen aus dem Monoid:

```
foldr f e xs = foldl f e xsy
```

Begründung Die Definitionen sind:

```
foldr            :: (a -> b -> b) -> b -> [a] -> b
foldr f z []      = z
foldr f z (x:xs)  = f x (foldr f z xs)

foldl            :: (a -> b -> a) -> a -> [b] -> a
foldl f z []      = z
foldl f z (x:xs)  = foldl f (f z x) xs
```

Mit Induktion:
Für die leere Liste gilt: `foldr f e [] =  e = foldl f e []`
Für eine Liste `[x]` gilt: `foldr f e [x] =  f x e = x =  f e x = foldl f e [x]`
Für eine Liste `x1:x2:xs` gilt: `foldr f e (x1:x2:xs)  = f x1 (f x2 (foldr f e xs))`. Assoziativität des Operators `f` ergibt: das ist gleich `f (f x1 x2) (foldr f e xs))` = `foldr f e (f x1 x2) : xs))`
Dieselbe Rechnung für `foldl` ergibt:
`foldl f e (x1:x2:xs)` = `foldl f (f (f e x1) x2) xs`. Assoziativität ergibt `foldl f  (f e (f x1 x2)) xs` = `foldl f  e (f x1 x2):xs`. Jetzt kann man Induktion über die Länge machen.

Bemerkung 9.28. Offenbar gilt, dass die Gleichung für unendliche Listen falsch sein kann, denn `foldl` ergibt immer bot für unendliche Listen, während `foldr` auch Ergebnisse liefern kann. Der bot-Fall gilt, aber die versteckte Annahme in der Induktion ist der Fall `[x]`, der gilt, aber man muss noch den Fall `x:bot` untersuchen. In diesem Falle erhält man: einerseits `f x bot`, andererseits `bot`. D.h. das erste Dualitätsgesetz für fold gilt für alle Listen, wenn der Operator f strikt im zweiten Argument ist: wenn `f x bot` = `bot` für alle x:
Die Induktion nach der Länge ist im Fall der unendlichen Listen dann auch erlaubt, da man eigentlich über die Partiallisten argumentiert.

Zweites Dualitätsgesetz für FOLD Seien f, g Operatoren, e eine Einheit, so dass

- `x 'f' (y 'g' z) = (x 'f'  y) 'g' z`
- `x 'f' e  = e 'g' x`

Dann gilt für alle endlichen Listen xs:

```
foldr f e xs = foldl g e xs
```

Zunächst muss man folgende Zwischenbehauptung für alle endlichen Listen xs zeigen:

```
x 'f' (foldl g y xs) = foldl g (x 'f' y) xs
```

Beweis:
Fall `[]`: Beide Seiten reduzieren auf `x 'f' y`
Fall `(z:zs)`: `x 'f' (foldl g y (z:zs))` = `x 'f' (foldl g (g y z) zs)`
= (Induktion) `foldl g (x 'f' (y 'g' z)) zs)`
= (Voraussetzung) `foldl g ((x 'f'  y) 'g' z) zs)`
Für die andere Seite erhält man:
`foldl g (x 'f' y) (z:zs)` = `foldl g ((x 'f' y) 'g' z) zs`
Die Zwischenbehauptung ist somit gezeigt.
Nachweis der eigentlichen Behauptung mit Induktion:

- Fall `[]`: `foldr f e [] = e = foldl g e []`
- Fall `x:xs`: `foldr f e (x:xs)` = `f x (foldr f e xs)`
  = (Induktion) `f x (foldl g e xs)`
  Die Zwischenbehauptung zeigt jetzt mit $y = e$ die Gleichung `f x (foldl g e xs)` = `foldl g (f x e) xs) =  foldl g (g e x) xs).`
  Andererseits: `foldl g e (x:xs)` = `foldl g (g e x) xs` und damit gilt die Behauptung des Dualitätssatzes.

Beispiel 9.29. Mit dem zweiten Dualitätssatz für fold kann man die Gleichheit der beiden `reverse`-Definition zeigen.

- `reverse1 = foldr snoc []  where snoc x xs = xs ++ [x]`
  Das entspricht der Definition

  ```
  reverse1  [] = []
  reverse1 (x:xs) = (reverse1 xs) ++ [x]
  ```

  was man leicht durch Induktion nachweisen kann (für alle Listen)
- `reverse2 = foldl cons []  where cons xs x = x:xs`
  Das entspricht der Definition

  ```
  reverse2 xs         = reverse2r xs []
  reverse2r []     ys  = ys
  reverse2r (x:xs) ys  = reverse2r xs (x:ys)
  ```

Um zu zeigen, dass die Ergebnisse gleich sind für endliche Listen, muss man nur nachweisen, dass `x 'snoc' (y 'cons' z) = (x 'snoc' y) 'cons' z` D.h. dass `(z:y) ++ [x] = z:(y ++ [x])]`. Das kann man mit einem Reduktionsschritt von `append` zeigen.
Man hat noch nachzuweisen, dass auch `x 'snoc' [] = [] 'cons' x`. Beides reduziert zu `[x]`.

Drittes Dualitätsgesetz für FOLD

Es gilt für alle endlichen Listen `xs`:

```
foldr f e xs = foldl (flip f) e (reverse xs)
```

Es gilt für alle `xs` (insbesondere alle Listen):

```
foldl f e xs = foldr (flip f) e (reverse xs)
```

Die Gleichungen sind konsistent mit der Intuition:
`foldr + 0 [1,2,3,4]` ergibt `1 + (2 + (3 + (4 + 0)))`, während `foldl (flip +) 0 [4,3,2,1]` ergibt `(((0 +' 4) +' 3) +' 2) +' 1` mit `flip + = +'`.

Nachweis
Zunächst eine andere Gleichung, die für alle Listen gilt, den Beweis kann man als Übungsaufgabe durchführen.

```
foldr g e (xs++ys) = foldr g (foldr g e ys) xs
```

Jetzt der Nachweis der zweiten Gleichung:

- Fall ungetypt. D.h. `xs` ist Abstraktion oder hat falschen Topkonstruktor: Beide Seiten sind äquivalent zu *bot*.
- Fall *bot*: Beide Seiten reduzieren zu *bot*.
- Fall `[]`: Beide Seiten reduzieren zu e.
- Fall `(x:xs)`.
  `foldl f e (x:xs)` $\rightarrow$ `foldl f (f e x) xs`
  `foldr (flip f) e (reverse (x:xs))`
  $\rightarrow$ `foldr (flip f) e ((reverse xs) ++ [x]))`
  = (ZWbeh) `foldr (flip f) (foldr (flip f) e [x]) (reverse xs)`
  = `foldr (flip f) (f e x) (reverse xs)`
  = (Induktion) `foldl f (f e x) xs`
  Damit sind die beiden Seiten gleich, und die Behauptung gilt für alle Listen.

Übungsaufgabe 9.30. Ist `foldr (:) [] = Id` ?

Beispiel 9.31. Den ersten Teil des Dualitätssatzes kann man verwenden, um `reverse (reverse xs) = xs` für endliche Listen zu zeigen:

```
foldr (:) [] xs = foldl (flip (:)) [] (reverse xs)
```

Die linke Seite wird zu `xs`, da `xs` endliche Liste. Auf der rechten Seite ist `foldl (flip (:)) [] = reverse`, deshalb ist die rechte Seite gerade `reverse (reverse xs)`.

Übungsaufgabe 9.32. Warum kann man mit der zweiten Aussage des Dualitätssatzes nicht `reverse (reverse xs) = xs` für alle (auch unendliche Listen) zeigen?

Verschmelzungs-Gesetze für Fold
Der Sinn und Zweck dieser Gesetze ist die Optimierung der Listenverarbeitung, die Zwischenlisten erzeugt und gleich wieder verwirft. Diese Zwischenlisten sollen vermieden werden durch automatische Umstellung der Verarbeitung.

FOLD-MAP Verschmelzung

```
foldr f a . map g = foldr (f.g) a
```

Das ist einfach nachzuweisen:

- Fall `bot`: ergibt `bot`.
- Fall `[]`: ergibt `a`.
- Fall `x:xs`:
  Linke Seite: `foldr f a . map g (x:xs)`
  = `foldr f a ((g x) : map g xs)` = `f (g x) (foldr f a (map g xs))`
  Die rechte Seite ergibt:
  `foldr (f.g) a (x:xs) = f (g x) (foldr (f.g) a xs)`.
  Anwenden der Induktionshypothese zeigt die Behauptung für den Fall `x:xs`.
- Danach kann man das Induktionsschema für Listen anwenden.

Die Optimierung besteht darin, die Verwendung von `map` zu eliminieren, denn die Ergebnisliste wird von `map` aufgebaut und von `foldr` gleich wieder abgebaut. Das Verschmelzungsgesetz für `foldl` und `map` ist:

```
foldl f a . map g  =  foldl (\x,y -> f x (g y))  a
```

Beispiel 9.33. Zu foldl-map-Verschmelzung. Das Ergebnis bei Anwendung auf eine Liste `[x1,x2,x3]` ist gerade `((a + (g x1)) + (g x2)) + (g x3)`

Nachweis

- Fall `bot`: Das Ergebnis ist jeweils bot.
- Fall `[]`: Das Ergebnis ist jeweils `[]`.
- Fall `x:xs`
  `foldl f a . map g (x:xs)`
  = `foldl f a ((g x): map g (x:xs))` = `foldl f (f a (g x)) (map g xs)`
  Auf der rechten Seite ergibt sich: `(foldl (\x,y -> f x (g y)) a (x:xs)`
  = `foldl (f.g) (f a (g x)) xs`. Anwenden der Induktionshypothese liefert die Behauptung in diesem Fall.

– Danach kann man das Induktionsschema für Listen anwenden.

FOLD-CONCAT Verschmelzung

```
foldr f a . concat = foldr (flip (foldr f)) a
```

Zum Nachweis dieser Gleichung benutzt man das Induktionsschema für Listen: Die Fälle `bot, []` sind einfach. Der Nachweis für `(xs:xss)` hat zwei Unterfälle

Fall `[]:xss`:
`foldr f a (concat ([]:xss))` = `foldr f a (concat (xss))`
`foldr (flip (foldr f)) a ([]:xss))`
= `((flip (foldr f)) []  (foldr (flip (foldr f)) a (xss)))`
= `foldr f (foldr (flip (foldr f)) a xss)  []`
= `foldr (flip (foldr f)) a  xss`

Fall `(x:xs):xss`:
`foldr f a (concat (x:xs):xss)` = `foldr f a (x: concat (xs:xss))`
= `f x (foldr f a (concat (xs:xss)))`
`foldr (flip (foldr f)) a ((x:xs):xss)`
= `(flip (foldr f)) (x:xs) (foldr (flip (foldr f)) a xss)`
= `foldr f (foldr (flip (foldr f)) a xss) (x:xs)`
Dazu zunächst noch die folgende Rechnung:
= `f x (foldr f (foldr (flip (foldr f)) a xss)  xs)`
`foldr (flip (foldr f)) a (xs:xss)`
= `(flip (foldr f)) xs (foldr (flip (foldr f)) a  xss)`
= `(foldr f (foldr (flip (foldr f)) a xss)  xs)`
Insgesamt hat man das Gesetz nachgewiesen.

Übungsaufgabe 9.34. Wie sieht das Gesetz für `foldl` aus?

```
foldl f a . concat = ???
```

FOLD-FILTER Verschmelzung

```
foldr f a (filter p xs) = foldr (\x y -> if (p x) then f x y else y) a xs

foldl f a (filter p xs) = foldl (\x y -> if (p y) then f x y else x) a xs
```

Den Nachweis kann man analog wie für die anderen Regeln führen.

Fold auf Bäumen: Verschmelzungsgesetze Die Faltung auf binären Bäumen kann man wie folgt definieren:

```
data Binbaum a = Blatt  a | Knoten (Binbaum a) (Binbaum a)
```

```
--  foldr Entsprechung    kann terminieren f"ur unendliche B"aume
foldrbt :: (a -> b -> b) -> b -> Binbaum a -> b
foldrbt op a (Blatt x) = op x a
foldrbt op a (Knoten x y) = (foldrbt op (foldrbt op a y) x)

--  foldl Entsprechung    terminiert nie f"ur unendliche B"aume
foldlbt :: (a -> b -> a) -> a -> Binbaum b -> a
foldlbt op a (Blatt x) = op a x
foldlbt op a (Knoten x y) = (foldlbt op (foldlbt op a x) y)

--  entspricht foldl mit Platzoptimierung:
foldlbtstr op a (Blatt x) = op a x
foldlbtstr op a (Knoten x y) = (strict (foldlbtstr op) (foldlbtstr op a x) y)

--  entspricht map:
mapbt f (Blatt x) = Blatt (f x)
mapbt f (Knoten bl br) =
                  Knoten (mapbt f bl)  (mapbt f br)

-- reverse-Entsprechung
reversebt (Blatt x) = (Blatt x)
reversebt (Knoten x y) = Knoten (reversebt y) (reversebt x)
```

Ein Verschmelzungsgesetz für `foldrbt` ist die Vermeidung des Baumaufbaus nach dem `map` über den Baum:

```
foldrbt op a . mapbt f    = foldrbt (\x y -> (op (f x) y)) a
```

Der Nachweis des Gesetzes für alle binären Bäume, wobei man ein auf binäre Bäume angepasstes Induktionsschema verwenden muss.

- Für bot ist das Ergebnis jeweils bot.
- Für `Blatt x` ist das Ergebnis jeweils `op (f x) a`.
- Für `Knoten x y` ergibt sich links: `foldrbt op a (mapbt f (Knoten x y))`
  $\rightarrow$ `foldrbt op a (Knoten (mapbt f x) (map f y))`
  `foldrbt op (foldrbt op a (mapbt f y))  (mapbt f x)`
  Es ergibt sich rechts:
  `foldrbt (\x y -> (op (f x) y)) a (Knoten x y)`
  $\rightarrow$ `foldrbt (\x y -> (op (f x) y))  (foldrbt (\x y -> (op (f x) y)) a y)  x`
  Jetzt kann man die Induktionshypothese anwenden.

Das Verschmelzungsgesetz für `foldlbt` ist:

```
foldlbt op a . mapbt f    = foldlbt (\x y -> (op x (f y))) a
```

Bei Kombinationen zwischen Listen und Bäumen kann man ebenfalls solche Verschmelzungen durchführen.

```
randbt = foldbt (:) []
```

Eine Verschmelzungsregel ist:

`map f . randbt` = `randbt (\x,y -> (f x):y)`

Der Beweis sei als Übungsaufgabe dem Leser überlassen.

Baum mit beliebig vielen Töchterknoten

Im folgenden der Quellcode mit Testfunktionen:

```
data Gbaum a = Gblatt  a | Gknoten [Gbaum a]

--  foldr Entsprechung   kann terminieren f"ur unendliche B"aume
foldrgt :: (a -> b -> b) -> b -> Gbaum a -> b
foldrgt op a (Gblatt x) = op x a
foldrgt op a (Gknoten []) = a
foldrgt op a (Gknoten (x:xs)) = (foldrgt op (foldrgt op a (Gknoten xs)) x)

testrgt = foldrgt (:) [] (Gknoten [Gknoten [Gblatt 1, Gblatt 2], Gblatt 3])
testrgt2 = foldrgt (:) [] (Gknoten (map Gblatt [1..]))

--  foldl Entsprechung   kann terminieren f"ur unendliche B"aume
foldlgt :: (a -> b -> a) -> a -> Gbaum b -> a
foldlgt op a (Gblatt x) = op a x
foldlgt op a (Gknoten []) = a
foldlgt op a (Gknoten (x:xs)) = (foldlgt op (foldlgt op a x) (Gknoten xs))

testlgt = foldlgt (flip (:)) [] (Gknoten [Gknoten [Gblatt 1, Gblatt 2], Gblatt 3])
testlgt2 = foldlgt (flip (:)) [] (Gknoten (map Gblatt [1..]))
--  testlgt2 = bot

--  entspricht  foldl mit Platzoptimierung:
foldlgtstr op a (Gblatt x) = op a x
foldlgtstr op a (Gknoten []) = a
foldlgtstr op a (Gknoten (x:xs)) =
               (strict (foldlgtstr op) (foldlgtstr op a x) (Gknoten xs))

testlgtstr = foldlgtstr (flip (:)) []
                        (Gknoten [Gknoten [Gblatt 1, Gblatt 2], Gblatt 3])
testlgtstr2 = foldlgtstr (flip (:)) []
                        (Gknoten (map Gblatt (take 10000 [1..])))

mapgt f (Gblatt a) = Gblatt (f a)
mapgt f (Gknoten (xs)) = Gknoten (map (mapgt f) xs)
testmapgt = mapgt quadrat (Gknoten [Gknoten [Gblatt 1, Gblatt 2], Gblatt 3])
```

```
-- Reverse
reversegt (Gblatt x) = (Gblatt x)
reversegt (Gknoten b) = Gknoten (foldl (\x y -> (reversegt y) : x) [] b)
```

Die Verallgemeinerung des fold-map Verschmelzungsgesetzes ist:

```
foldrgt op a . mapgt f    = foldrgt (\x y -> (op (f x) y)) a
```

Dies ergibt für `Gblatt a` gerade `Gblatt (f a)`.
Für `(Gknoten [])`: Beide Seiten ergeben a.

Für `(Gknoten (x:xs))`:
`foldrgt op a (mapgt f (Gknoten (x:xs)))` $\rightarrow$
`foldrgt op a (Gknoten (map (mapgt f) (x:xs)))` $\rightarrow$
`foldrgt op a (Gknoten ((mapgt f  x) : map (mapgt f) xs))` $\rightarrow$
`(foldrgt op (foldrgt op a (Gknoten (map (mapgt f) xs))) (mapgt f  x))`.
Die rechte Seite ergibt: `foldrgt (\x y -> (op (f x) y)) a (Gknoten (x:xs))`
$\rightarrow$ `foldrgt (\x y -> (op (f x) y)) a (Gknoten (x:xs))` $\rightarrow$
`foldrgt (\x y -> (op (f x) y))   (foldrgt (\x y -> (op (f x) y)) a (Gknoten xs))  x`
Jetzt kann man ein für `Gbaum` angepasstes, verallgemeinertes Induktionsschema verwenden.

Bemerkung 9.35. Das Prädikat "terminiert" kann man in stetigen zusammengesetzten Prädikaten verwenden: z.B. das Prädikat

$P(x) \equiv$ `(f x)` terminiert

kann man in modifizierter Form durchaus verwenden, wenn man es als `(f x)` $>_c bot$ darstellt. Denn wenn für alle i gilt: $s_i >_c bot$, dann gilt auch $club_i(s_i) >_c bot$. Damit das brauchbar ist, muss man evtl. den Anfang der Kette weglassen.

9.3 Nachweis von Aussagen in typisierten funktionalen Programmiersprachen

In diesem Paragraphen nehmen wir an, dass alle Ausdrücke und Unterausdrücke polymorph getypt sind. Die Grundausdrücke vom Grundtyp[2] α bezeichnen wir mit $Exp(\alpha)$. Wir nehmen an, dass die Typisierung die folgenden Eigenschaften erfüllt:

- Wenn t ein geschlossener Ausdruck ist, α ein Grundtyp, $t :: \alpha$, und $t \rightarrow t'$, dann auch $t' :: \alpha$.
- Wenn $t :: \tau \equiv (c\ \alpha_1 \ldots \alpha_n)$, wobei c ein Typkonstruktor ist, aber nicht gleich $\rightarrow$, dann gilt einer der Fälle:
  - t hat keine WHNF, oder

[2] ein Grundtyp ist ein Typ ohne Variablen

- t hat eine WHNF der Form $c_i\ t_1 \ldots t_m$, und c_i ist ein Datenkonstruktor zum Typ c, und der Typ von t_j ist eindeutig bestimmbar. D.h. es gibt eine Funktion $\mathtt{comptyp}(\tau, c_i, j)$, die einen Grundtyp liefert, und es gilt $t_j :: \mathtt{comptyp}(\tau, c_i, j)$.

– Wenn t geschlossen ist, und $t :: \alpha_1 \to \alpha_2$, dann hat t entweder keine WHNF oder eine FWHNF.

Jetzt können wir das Induktionsschema angeben:

Induktionsschema für stetige Prädikate unter Typisierung:

– Zeige $P(\bot)$.
– Wenn $\alpha = \tau_1 \to \tau_2$ ein Funktionstyp, dann zeige $P(\lambda x : \tau_1.r :: \tau_2)$.
– Wenn $\alpha = (c\ \tau_1 \ldots \tau_n)$: Zeige für alle Konstruktoren c_j, die zum Typ c gehören:
$P(x_1 :: \mathtt{comptyp}(\alpha, c_j, 1)) \wedge \ldots \wedge P(x_1 :: \mathtt{comptyp}(\alpha, c_j, ar(c_j))$
$\Rightarrow P(c_j\ x_1 \ldots x_{ar(c_j)}))$

Dann kann man schließen: $P(s)$ gilt für alle Typen α und alle Ausdrücke $s : \alpha$.

Die Begründung für dieses Induktionsschema ist analog wie für unendliche Listen bzw. für alle Ausdrücke: Für Ausdrücke, die aus endlich vielen Konstruktoren, bot und FWHNFs aufgebaut sind, kann man die Eigenschaft dann mit Induktion begründen. Für die Ausdrücke, die man nur "unendlich" darstellen kann mittels Konstruktoren, bot und FWHNFs, verwendet man Stetigkeit. Die Aussage für alle Ausdrücke erhält man über die clubs von Partialausdrücken analog zu Partiallisten. Der entsprechende Beweis ist analog zu Lemma 9.15.

10 Typsysteme, Typüberprüfung

Wir haben bisher algebraische Datentypen in KFP und Haskell kennengelernt, und gesehen, dass es in KFP dynamische Typfehler geben kann, und dass Haskell und ML ein strenges Typsystem haben, so dass es keine Typfehler zur Laufzeit gibt. Wir werden ein polymorphes Typsystem vorstellen, und zwei Varianten des dazugehörigen Typüberprüfungsalgorithmus (Hindley, Milner, Mycroft, Abramsky, Damas).
Wir legen die Sprache KFP+ mit der Erweiterung bezüglich der Typangaben der Konstruktoren zugrunde, so dass für jeden Konstruktor im Deklarationsteil ein Typ mit angegeben werden muß.
Die Typregeln, die wir angeben werden, können analog auch auf andere funktionale Programmiersprachen angewendet werden, wobei auch für kompliziertere Konstrukte Typregeln entworfen werden können.

Erwünschte Eigenschaften eines Typsystems:

Wenn ein Ausdruck positiv überprüft worden ist, d.h. einen Typ hat, dann sollte nach der Kompilierung zur Laufzeit kein dynamischer Typfehler auftreten. Außerdem sollten keine zu starken Programmierbeschränkungen durch das Typsystem erzwungen werden.
Typisierung sollte effizient entscheidbar sein.

Das Milner-Typsystem ist ein starkes und auch statisches Typsystem, d.h. jeder erlaubte Ausdruck (und Unterausdruck) muß einen Typ haben. Die Typen existieren nur zur Compilezeit. Das Laufzeitsystem hat (fast) keine Typinformation mehr. Das ist im Gegensatz zu dynamischen Typsystemen, bei denen Typen auch zur Laufzeit der Programme geprüft werden.
Man sieht schnell ein, dass es kein Typsystem geben kann, das alle KFP+-Programme erkennt, die keine Typfehler machen werden. Denn:

Satz 10.1. *Es gibt keinen Algorithmus, der für jedes eingegebene KFP+-Programm entscheidet, ob es einen Typfehler zur Laufzeit geben wird oder nicht.*

Beweis. Die Zurückführung auf das Halteproblem ist einfach: Betrachte den Ausdruck `(if t then (case_Bool Nil Nil Nil) else (case_Bool Nil Nil Nil))`. Dieser Ausdruck ergibt bei beliebigem t (das keine Typfehler erzeugt) genau dann einen Typfehler, wenn t terminiert. Da man in KFP eine (getypte) Turingmaschine codieren kann (benutze t), hat man das Halteproblem zu entscheiden. □

Das Typsystem kann Ausdrücke als ungetypt zurückweisen, d.h. es werden bestimmte Programme (die ungetypten) verworfen und nicht akzeptiert. Die Konsequenz aus der Unentscheidbarkeit ist jedoch, dass es in jedem Typsystem ungetypte Programme gibt, die sinnvoll sind, bzw. die zumindest keine dynamischen Typfehler provozieren. Typüberprüfung kann zur Verbesserung der Effizienz beitragen, da i.a. nach der Typüberprüfung keine dynamische Typprüfungen zur Laufzeit mehr notwendig sind.

10.1 Polymorphes Typsystem nach Milner (& Mycroft)

Syntax für Typ-Ausdrücke:
Typ ::= Variable | (Typkonstruktor $Typ \ldots Typ$) | $Typ \rightarrow Typ$

(Typ-)Variable: bedeutet beliebiger Typ, Bezeichnung: $\alpha, \beta, \gamma, \delta, \ldots$. Gleiche Variablen bedeuten gleichen Typ.
Typkonstruktor: zu jedem algebraischen Datentyp gibt es einen. Die Stelligkeit wird bei der Definition der Daten-Konstruktoren festgelegt: Das ist die Anzahl der verschiedenen Typvariablen in den Konstruktoren. z.B. bei `List` (bzw. $[\cdot]$) ist es eine Typvariable (der Typ der Elemente). Bei `Paar` (bzw. $(\cdot, \cdot)$) sind es zwei Typvariablen.
$\rightarrow$ Typ-Konstruktor für Funktionen.

Beachte die Klammer-konventionen zu Typen.

Beispiel 10.2.

- Typ von $*$: `Int` $\rightarrow$ `Int` $\rightarrow$ `Int`.
- Typ von `Id`: $\alpha \rightarrow \alpha$.
- Typ des Paar-Konstruktors: `Paar`: $\alpha \rightarrow \beta \rightarrow (\alpha, \beta)$.

Hier bedeuten Vorkommen der gleichen Typvariable auch, dass dort immer Objekte des gleichen Typs gemeint sind: `Id` nimmt Objekte vom Typ α und gibt als Resultat wieder solche vom Typ α zurück.

- Typen, die Typvariablen enthalten, nennen wir auch *polymorph*;
- Typen ohne Variablen nennen wir *monomorph* oder *Grundtypen*.

Die Bedeutung des Typs $\alpha \rightarrow [\alpha] \rightarrow [\alpha]$ kann man auf zwei Arten interpretieren:

1. α ist allquantifiziert . D.h. der Typ ist: $\forall \alpha.\alpha \rightarrow [\alpha] \rightarrow [\alpha]$.
2. Als eine Beschreibung einer Menge von Grundtypen, die ein Ausdruck haben kann. In diesem Falle ist das die Menge $\{\tau \rightarrow [\tau] \rightarrow [\tau] \mid \tau \text{ ist ein Grundtyp}\}$.

Annahme Damit wir eine Basis für konkrete Typen von Ausdrücken haben, wird davon ausgegangen, dass die Datenkonstruktoren einen vorgegebenen Typ haben, der bei der Deklaration der algebraischen Datentypen in einem Programm mitangegeben werden muß. Dieser Typ gibt die Stelligkeit an, und die möglichen Typen und Typkombinationen der Argumente. Der Zieltyp muß von der Form $T\ \alpha_1 \ldots \alpha_n$ sein, wobei T jeweils der Typkonstruktor des zugehörigen algebraischen Datentyps ist und α_i Typvariablen sind. Alle Typvariablen, die im Typ der Argumente vorkommen, sollten auch im Zieltyp des Datenkonstruktors vorkommen.

Beispiel 10.3.

Paare: $(.,.) : \alpha \rightarrow \beta \rightarrow (\alpha, \beta)$

Listen: `List` (bzw. []) wird als einstelliger Typkonstruktor definiert.
`Nil` und : sind die (Daten-) Konstruktoren. `Nil`: $[\alpha]$ und ':' : $\alpha \rightarrow [\alpha] \rightarrow [\alpha]$
Dies definiert Listen, bei denen die Elemente den gleichen Typ haben. `(1 : (True : Nil))` ist dann nicht mehr möglich, da das zweite Argument von `:` einen Listentyp haben muß.

Die Konstruktoren von `Bool` haben folgende Typangaben: `True`: `Bool` und `False`: `Bool`.

Beispiel 10.4. markierte binäre Bäume:

- `Tree` ist der einstellige Typkonstruktor.
- `Node,Leaf` sind die Konstruktoren.
- `Node`: (`Tree` α) $\rightarrow$ (`Tree` α) $\rightarrow$ (`Tree` α)
- `Leaf`: $\alpha \rightarrow$ (`Tree` α)

Instanz eines Typs: Dazu benötigt man (Typ-)Substitutionen σ: das sind Abbildung, die Variablen Typterme zuordnen. Diese kann man dann auch auf komplexere Typterme anwenden durch die Regel $\sigma(\mathtt{c}\ t_1 \dots t_n)) = (\mathtt{c}\ (\sigma t_1) \dots (\sigma t_n))$ Dann ist $\sigma(\tau)$ Instanz eines Typs τ. Wenn nach der Instanziierung der Typ keine Variablen mehr enthält, dann sprechen wir von *Grundinstanz*.
z.B. Instanzen von $[\alpha]$ sind [`Int`] und $[[\alpha]]$, wobei [`Int`] eine Grundinstanz ist.

10.2 Typregeln (auf Grundtypen) für KFP

Wir geben Typregeln an, wobei τ jeweils ein Grundtyp ohne Variablen ist.
Die Regeln sind folgendermaßen aufgebaut:

$$\frac{\text{Annahmen}}{\text{neue Folgerung}}$$

Oberhalb der Linie stehen die Annahmen, die von der Form "Ausdruck t ist von Typ τ" sind, und unterhalb steht der neu zu schließende Typ für einen Ausdruck. Regeln ohne Annahmen nennt man auch **Axiome**. Bei diesen wird die Annahme und der Strich weggelassen.
Die Annahmen über die möglichen Typen der freien Variablen und der Konstanten sind i.a. in einer Menge gesammelt. Von den Variablen nehmen wir an, dass diese genau eine Typannahme $x : \tau$ haben. Die Konstanten können mehrere Typannahmen haben: $c : \tau_1, \dots, c : \tau_n$. Im allgemeinen ist dies eine Menge T von Grundinstanzen eines einzigen Typs mit Variablen. In diesem Fall schreiben wir diese Menge als einen Typ mit Variablen $c : \forall(\tau)$. Diese repräsentiert $T = \{\tau' \mid \tau'$ ist Grundinstanz von $\tau\}$. In diesem Fall ist die Menge der zugehörigen Grundtypen i.a. unendlich groß.
Die Regeln definieren eine Herleitbarkeitsrelation $\vdash$ zwischen Annahmen über Typen von Variablen und Konstanten. $A \vdash s : \tau$ bedeutet, dass aus den Annahmen A über die Typen der Variablen und Konstanten in s das Typsystem den Typ τ von s herleiten kann. Da man für Konstanten mehrere Typannahmen machen kann, ergibt sich somit auch, dass man möglicherweise mehrere Typinferenzen ziehen kann. Damit kann man i.a. eine Menge von Typen für einen Term herleiten.
Das Semikolon trennt verschiedene mit "und" verbundene Annahmen über die Ableitbarkeit von Typen.

10.3 Typisierungsregeln

Axiom für Konstanten (Mengenschreibweise)

$$A \cup \{c : T\} \vdash c : \tau \text{ wenn } \tau \in T$$

Axiom für Konstanten (polymorphe Typen)

$$A \cup \{c : \forall(\tau)\} \vdash c : \tau' \text{ wenn } \tau' \text{ Grundinstanz von } \tau$$

Axiome für freie Variablen

$$A \cup \{x : \tau\} \vdash x : \tau$$

Anwendungsregel:

$$\frac{A \vdash a : (\tau_1 \to \tau_2); A \vdash b : \tau_1}{A \vdash (a\ b) : \tau_2}$$

Einsetzungsregel (Superkombinatorregel)

$$\frac{\begin{array}{l} A \cup \{x_1 : \tau_1, \ldots x_n : \tau_n\} \vdash e : \tau; \\ \texttt{sc}\ x_1 \ldots x_n = e \text{ ist die Definition für } sc \end{array}}{A \vdash sc : \tau_1 \to \ldots \to \tau_n \to \tau}$$

Nebenbedingung ist, dass A keine Annahmen über x_i hat.

Abstraktion:

$$\frac{A \cup \{x : \tau_1\} \vdash t : \tau_2}{A \vdash (\lambda x.t) : \tau_1 \to \tau_2}$$

case-Regel:

$$\frac{\begin{array}{l} A \vdash e : \alpha \\ A \cup \{x_{1,1} : \tau_{1,1} \ldots x_{1,n(1)} : \tau_{1,n(1)}\} \vdash (c_1\ x_{1,1} \ldots x_{1,n(1)}) : \alpha \\ \ldots \\ A \cup \{x_{m,1} : \tau_{m,1} \ldots x_{m,n(m)} : \tau_{m,n(m)}\} \vdash (c_m\ x_{m,1} \ldots x_{m,n(m)}) : \alpha \\ A \cup \{x_{1,1} : \tau_{1,1} \ldots x_{1,n(1)} : \tau_{1,n(1)}\} \vdash e_1 : \tau \\ \ldots \\ A \cup \{x_{m,1} : \tau_{m,1} \ldots x_{m,n(m)} : \tau_{m,n(m)}\} \vdash e_m : \tau \\ \text{und } \alpha \text{ hat Top-Typkonstruktur, der zu } D \text{ gehört} \end{array}}{A \vdash (\texttt{case}_D\ e\ ((c_1\ x_{1,1} \ldots x_{1,n(1)}) \to e_1) \ldots ((c_m\ x_{m,1} \ldots x_{m,n(1)}) \to e_m)) : \tau}$$

Mit diesen Regeln kann man bereits Typen für nicht rekursiv definierte Superkombinatoren erschließen. Man kann oft für Superkombinatoren polymorphe Typen angeben, die alle herleitbaren Grundtypen repräsentieren. Wir geben dafür jedoch keine Regeln an, da das im Typsystem für Typen mit Typvariablen besser und verständlicher gemacht werden kann.
Ein Term t ist *wohlgetypt*, wenn man für t einen Typ herleiten kann. Das ist offenbar nur dann möglich, wenn auch alle Unterausdrücke einen Typ haben.

Beispiel 10.5. `unit-list x = cons x nil`
Sei $A = \{x : \tau, \forall(\texttt{Nil} : [\beta]), \texttt{Cons} : \forall(\alpha \to [\alpha] \to [\alpha])\}$ die Menge der Annahmen. Beachte, dass $[\beta]$ hier für die Menge aller Grundinstanzen steht, entsprechend bei `Cons`. Hierbei ist τ ein (unbekannter) Grundtyp.
Aus dem Typ von `Cons` ergibt sich, dass zunächst `(cons x)` den Typ $[\tau] \to [\tau]$ hat. Weitere Anwendung auf `Nil` erfordert, dass der Typ des Arguments und der

erforderliche Typ des Funktionsausdrucks übereinstimmen. Da $[\tau]$ eine Instanz von $[\beta]$ ist, erhalten wir für das Ergebnis $[\tau]$.
D.h. für alle Grundtypen τ gilt: `unit-list`: $\tau \to [\tau]$. Da das für alle Grundtypen τ gilt, ist der Typ von `unit-list` somit eine Menge von Grundtypen, die man mit $\alpha \to [\alpha]$ beschreiben kann.

Beispiel 10.6. Die Funktionen `twice` sei definiert als: `twice f x = f ( f x)`
Die Typannahme sei $A = \{x : \tau_1, \texttt{f} : \tau_2\}$.
Der Typ von `(f x)` muß existieren: deshalb muß $\texttt{f} : \tau_2 = \tau_1 \to \tau_3$ sein, wobei τ_3 noch unbestimmt ist, aber der Typ von `(f x)` muß existieren: `(f x)`: τ_3. Dies bedeutet eigentlich, dass nur Annahmen folgender Form sinnvoll sind: $A' = \{x : \tau_1, \texttt{f} : \tau_1 \to \tau_3\}$.
Der Typ von `f (f x)` muß existieren: Wir haben angenommen, dass Argumentvariablen nur einen einzigen Typ haben, der in die Typberechnung eingeht: Dann ist $\tau_1 = \tau_3$ und `f (f x)`: τ_1. Da wir die möglichen Annahmen weiter eingeschränkt haben, gilt jetzt: $A'' = \{x : \tau_1, \texttt{f} : \tau_1 \to \tau_1\}$ und daraus läßt sich jetzt `f (f x)`: τ_1 herleiten. Damit erhalten wir als Typ von `twice` mit der Ersetzungsregel: $(\tau \to \tau) \to \tau \to \tau$. Da dies ohne Annahmen über `twice` hergeleitet werden konnte, und auch für alle Grundtypen τ hergeleitet werden kann, kann man als Typ von `twice` $(\alpha \to \alpha) \to \alpha \to \alpha$ angeben.

Betrachtet man die Annahmen genauer, wird klar, dass man statt einer unendlichen Menge von verschiedenen Annahmen für die freien Variablen, wobei man für jede Kombination dieser Annahmen jeweils eine Typherleitung gemacht werden muß, diese auch mit Typvariablen versehen kann.
Die Annahme, dass man für die Eingabeargumente nur einen Grundtyp zuläßt (statt einer Menge), ist wesentlich für die Einfachheit des Typsystems. Das ist die sogenannte *Monomorphie-annahme*. Man kann sich auch Typsysteme vorstellen, die für Funktionen als Argumente verschiedene Typinstanzen dieser Funktionen im Rumpf verwenden. Als Beispiel wollen wir die Funktion `selfapply` betrachten, deren Rumpf als Unterausdruck in der Definition des Fixpunktkombinators Y vorkommt:

Beispiel 10.7. Berechne den Typ von `selfapply x = x x` nach zwei verschiedenen Methoden.

1. Argumente dürfen nur mit einem Typ angewendet werden (sogenannte Monomorphie-Einschränkung):
   Annahme: $\{\texttt{selfapply} : \alpha, x : \tau\}$
   `(x x)` hat einen Typ, deshalb gilt $\tau = \tau_1 \to \tau_2$ und $\tau_1 = \tau$. Dies ergibt einen Fehler, denn es gibt keine Grundtypen, die obige Annahme erfüllen. Aber: es gibt Argumente, so dass `(selfapply x)` vernünftig ist, Z.B: `selfapply` $I \to (I\ I) \to I$.
2. Argumente dürfen eine Menge von Typen haben: $\{\texttt{selfapply} : S; x : T\}$
   `(x x)` muß einen Typ haben $\Rightarrow T$ muß folgende Eigenschaft haben:
   (‡) Es gibt Typen $\tau_1 \in T$ und $\tau_1 \to \tau_2 \in T$.
   Der von T abgeleitete Typ von `(x x)` ist dann: $T_1 = \{\tau_2 \mid$ Es gibt

$\tau_1 \to \tau_2 \in T$ und $\tau_1 \in T\}$
Die Typisierung von `selfapply` kann dann nicht mehr als Menge von Typen beschrieben werden, die Instanz eines polymorphen Typausdrucks ist, sondern nur noch als komplexere Bedingung an Mengen von Grundtypen.

Bemerkung Die Typisierung beschreiben wir für die Sprache KFP+, da es in KFP Ausdrücke gibt, die wir benötigen, z.B. Y, die aber keinen polymorphen Typ haben. In KFP+ kann man durch die rekursive Definitionsmöglichkeit von Superkombinatoren dieses Problem verdecken, so dass man auch für rekursive Superkombinatoren einen polymorphen Typ erhält.

10.4 Typen für rekursive Superkombinatoren

Unser bisheriges Typsystem kann noch keinen Typ für rekursiv definierte Superkombinatoren erschließen, da man den Typ eines rekursiven Superkombinators bereits in die Voraussetzung A aufnehmen muß.

Definition 10.8. *(Fixpunkt, konsistente Annahme) Sei A eine Menge von (nichtleeren) Annahmen über alle Konstruktoren und definierte Superkombinatoren sc_i, wobei $A = sc_1 : S_1, \ldots, sc_n : S_n$. Wenn für alle $\tau_i \in S_i$ und für alle i: $A \vdash sc_i : \tau_i$, wobei die letzte angewendete Regel jeweils die Superkombinatorregel ist, dann ist A eine* konsistente Typannahme *für alle Superkombinatoren $sc_i, i = 1, \ldots, n$.*

Wir erlauben nur eine Annahme für jeden Superkombinator, wobei diese Annahme sich als polymorpher Typ ausdrücken lassen muß.
Die Gleichheit läßt sich dann leicht testen, denn für polymorphe Typen mit Variablen gilt:

> τ_1 repräsentiert die gleiche Menge von Grundinstanzen wie τ_2, wenn τ_1 und τ_2 bis auf Umbenennung von Variablen gleich sind.

Analog repräsentiert τ_2 mehr Typen als τ_1, wenn $\sigma(\tau_2) = \tau_1$ ist für eine Typsubstitution σ (mit Variablen). Dieser Test ist in linearer Zeit durchführbar.
Wenn man eine konsistente Typannahme hat, kann man die Typannahme als Typ der entsprechenden Superkombinatoren ansehen, wenn ein möglichst allgemeiner Typ angenommen wurde.

Definition 10.9. *Ein Programm ist* wohlgetypt*, wenn es eine konsistente Typannahme für alle Superkombinatoren gibt und der auszuwertende Ausdruck einen polymorphen Typ hat bezüglich dieser Typannahmen.*

Bevor wir zu einem Typalgorithmus kommen, der in den meisten Fällen allgemeinste Typen berechnet, argumentieren wir, dass das obige Typsystem garantiert, dass keine dynamischen Typfehler gemacht werden können, wenn die Programme wohlgetypt sind.

Beispiel 10.10. Typisierung eines rekursiven Kombinators.
`repeat x = Cons x (repeat x)`
$A = \{\texttt{repeat} : \forall\alpha, \texttt{Cons} : \forall(\beta \to [\beta] \to [\beta]), x : \tau\}$
`(repeat x)` ist getypt, deshalb hat es den Typ $\tau \to \alpha$
`(cons  x (repeat x))` ist getypt, deshalb $[\tau] = \alpha$
und damit `(cons  x (repeat x))`: $[\tau]$.
Das Ergebnis in der Typschreibweise mit neuen Variablen: `repeat`: $\alpha \to [\alpha]$

Wir testen die neuen Annahmen auf Konsistenz:
$A_1 = \{x : \tau, \texttt{repeat} : \forall(\alpha \to [\alpha]), \texttt{Cons} : \beta \to [\beta] \to [\beta]\}$
`(repeat x)`: $[\tau]$
`(Cons x (repeat x))`: $[\alpha]$ und $\beta = \alpha = \tau$
Der Typ ist damit wieder $\tau \to [\tau]$ für alle τ, wobei dies mit der Superkombinatorregel herleitbar war. Damit ist A_1 eine konsistente Typannahme.

Wir wollen jetzt anhand des Typinferenzsystems für die Grundtypen begründen, dass wohlgetypte Programme keine dynamischen Typfehler machen.

Lemma 10.11. *Für direkt dynamisch ungetypte Ausdrücke läßt sich kein Typ herleiten.*

Begründung. Es gibt zwei Fälle, in denen ein Ausdruck dynamisch ungetypt ist: Wir können der Einfachheit halber annehmen, dass alle Unterausdrücke einen Typ haben.

1. `(case_T e ...)` und `e` ist in WHNF, aber nicht von der richtigen Form. Wenn e eine Konstruktor-WHNF mit zuwenig Argumenten ist, dann ist der Typ immer von der Form $\alpha \to \beta$. Dies paßt nicht zur case-Regel. Analog gilt: wenn e eine Kombinator-WHNF mit zuwenig Argumenten ist, dann ist der Typ ebenfalls von der Form $\alpha \to \beta$.
2. e ist eine Anwendung eines Konstruktor auf zuviele Argumente $(c\, t_1 \ldots t_m)$. Da die Anwendung auf genügende viele Argumente einen Typ der Form $(K\ t_1 \ldots t_n)$ hat, kann dies kein Funktionstyp sein, und somit ist $(c\, t_1 \ldots\ t_n)\ t_{n+1}$ nicht typisierbar. □

Lemma 10.12. *(subject reduction) Angenommen, wir haben eine konsistente Typannahme für alle Superkombinatoren. Dann gilt: Wenn $t \to t'$ eine Reduktion ist und man für t den Grundtyp τ herleiten kann, dann kann man auch für t' den Typ τ herleiten.*

Beweis. Wir betrachten die Fälle der verschiedenen Reduktionen: Es genügt, einen Redex zu betrachten. Bei Betrachtung der Oberterme genügt es, sich davon zu überzeugen, dass nur die Typen, nicht aber die Struktur der Terme eine Rolle spielt.

1. $sc\ t_1 \ldots t_n \to e[t_1/x_1 \ldots t_n/x_n]$ und $sc\ x_1 \ldots x_n = e$ sei die Definition für sc. Vor der Reduktion haben wir unter einer Annahme A die Typen t_1 :

$\tau_1, \ldots, t_n : \tau_n, (sc\ t_1 \ldots t_n) : \tau$ hergeleitet, und $sc : \tau_1 \to \ldots \to \tau_n \to \tau \in A$. Da die Typannahmen konsistent sind, und die Typisierungsregeln nur den Typ der Terme, nicht aber deren innere Struktur beachten, erhalten wir, dass für $e[t_1/x_1 \ldots t_n/x_n]$ auch der Typ τ herleitbar ist.

2.

$$(\texttt{case}_A\ (c_i\ t_1 \ldots t_{n(i)})) \quad ((c_1\ x_{1,1}, \ldots x_{1,n(1)}) \to exp_1) \ldots ((c_m x_{m,1}, \ldots x_{m,n(m)}) \to exp_m)$$

reduziert zu:

$$exp_i[t_1/x_{i,1}, \ldots, t_{n(i)}/x_{i,n(i)}]$$

Die Typregel für `case` erzwingt, dass die Typen für alle Ausdrücke $(c_i\ t_1 \ldots t_{n(i)})$ und $(c_j\ x_{j,1} \ldots x_{j,n(j)})$ gleich sind und als obersten Typkonstruktor K_A haben. Wegen der Konvention über die Typvariablen in der Definition von Datenkonstruktoren zu einem Typ gilt, dass die Typen der Argumente von $(c_i\ t_1 \ldots t_n(i))$ und $(c_i\ x_{i,1}, \ldots x_{i,n(i)})$ sich aus dem gemeinsamen Typ wieder eindeutig bestimmen lassen, so dass für alle j die Typen von t_j und $x_{i,j}$ die gleichen sind. Somit stimmt der herleitbare Typ des reduzierten Ausdrucks mit dem des `case` überein.

Damit ist jeder Typ des Ausdruck vorher auch noch Typ des Ausdrucks nach Reduktion, insbesondere, wenn der Ausdruck in WHNF (oder Normalform) ist. Außerdem sind alle Unterausdrücke jedes Terms innerhalb der Reduktion getypt. □

Lemma 10.13. *Für dynamisch ungetypte Ausdrücke läßt sich kein Typ herleiten.*

Beweis. Folgt aus Lemma 10.11 und aus Lemma 10.12. Erinnerung: dynamisch ungetypte Ausdrücke sind solche, die mit evtl. mehreren Normalordnungsreduktion auf einen direkt ungetypten Ausdruck reduzieren. □

Damit gilt der Satz:

Satz 10.14. *: (Milner) Wohlgetypte Programme machen keine dynamischen Typfehler.*

Bemerkung: Dieser Satz ist etwas allgemeiner als für das Milnersche Typsystem, da in Milners System eine etwas eingeschränktere Typberechnungsmethode verwendet wird.

11 Berechnung von allgemeinsten Typen von Ausdrücken.

Man möchte natürlich möglichst den allgemeinsten Typ von Superkombinatoren berechnen, da dieser die besten Verwendungsmöglichkeiten erlaubt.

Konzeptuell kann man dies, indem man zunächst die allgemeinsten Annahmen macht, und dann in einer Fixpunktiteration diese Annahmen solange verkleinert, bis die Eigenschaft erfüllt ist. Diese Fixpunktiteration terminiert möglicherweise nicht, und somit wird auch nicht klar, ob der Ausdruck typisierbar ist. Die Nichtterminierung könnte auch bedeuten, dass es keinen Typ für den betrachteten Superkombinator gibt.
Im folgenden beschreiben wir einen solchen Algorithmus. Diese Typisierungsalgorithmen haben natürlich auch Rückwirkungen auf den Programmierstil, denn nur wohlgetypte Programme werden akzeptiert.
Wir schreiben die Annahmen, die Typvariablen enthalten, mit einem Präfix $\forall$, um anzudeuten, dass die Typvariablen unter einem Quantor stehen und somit deren Name keine Rolle spielt.

Beispiel 11.1. Versuche binäre Bäume als verschachtelte Listen darzustellen. Definiere die Funktion `concat`, die alle Blätter in einer Liste sammelt. Kann man einen Typ für diese Funktion angeben?

11.1 Unifikation

Das bisherige Verfahren ist noch kein vernünftiger Typcheck-Algorithmus, aber weist schon darauf hin, was man hinzufügen muß, um einen Typinferenzalgorithmus zu bekommen. Wir fügen nur eine bessere Behandlung der Behandlung der Gleichungen, d.h. der Beschränkungen der Annahmen durch.
Dieser Algorithmus benötigt Unifikation (Gleichungslösen) von polymorphen Typen, um mit diesen Beschränkungen umzugehen.
Unifikation operiert auf einer Menge von Gleichungen, wobei wir $\doteq$ als symmetrisch betrachten. Die Regeln operieren auf den Gleichungen zwischen Typen und verändern die Menge der Gleichungen. Die Regeln sind nichtdeterministisch formuliert und können in irgendeiner Reihenfolge angewendet werden.

1. $(c\ldots) \doteq (d\ldots)$ dann FAIL, wenn c und d verschiedene Typkonstruktoren sind.
2. $(c\, t_1 \ldots t_n) \doteq (c\, s_1 \ldots s_n)$: dann lösche diese Gleichung und ersetze sie durch $s_1 \doteq t_1, \ldots, s_n \doteq t_n$.
3. $\alpha \doteq \tau$ und α ist Typvariable, $\tau \neq \alpha$, α kommt in τ nicht vor und α kommt nochmal in den Gleichungen vor. Dann ersetze überall in den anderen Gleichungen α durch τ.
4. $\alpha \doteq \alpha$. Dann lösche diese Gleichung
5. $\alpha \doteq \tau$, wobei α Typvariable, $\tau \neq \alpha$ und α kommt in τ vor, dann FAIL (sogenannter Occurs-check, erscheint als “infinite type“ in Fehlermeldungen)

Eigenschaften der Unifikation:

1. das Ergebnis ist bis auf Umbenennung der Variablen eindeutig
2. Der Algorithmus ist total korrekt. Er terminiert und findet eine Lösung, wenn eine existiert.

Bei naiver (ungünstiger) Implementierung ist der Algorithmus zeit-exponentiell. Er kann aber quadratisch, in dieser Variante als $O(n * log(n))$ und in einer anderen Variante sogar linear implementiert werden.
Wir sagen, dass ein Gleichungssystem *gelöst* ist, wenn es von der Form $\{\alpha_1 \doteq \tau_1, \ldots \alpha_n \doteq \tau_n\}$ ist, und die Variablen α_i alle verschieden sind und nicht in den τ_j vorkommen. Wenn ein Gleichungssystem gelöst ist, kann man eine Substitution ablesen: Aus $\{\alpha_1 \doteq \tau_1, \ldots, \alpha_n \doteq \tau_n\}$ kann man $\{\alpha_1 \to \tau_1, \ldots, \alpha_n \to \tau_n\}$ ablesen.

11.2 Der iterative Typisierungsalgorithmus

Er soll die bestmöglichen (d.h. allgemeinsten) polymorphen Typen für die Superkombinatoren berechnen, und ebenso den Typ aller Unterausdrücke und natürlich auch des auszuwertenden Ausdrucks.
Normalerweise macht man zuerst eine Abhängigkeitsanalyse zwischen den Superkombinatoren und startet die Typisierung von unten, d.h. mit den Superkombinatoren, die von den wenigsten anderen abhängen. Danach kann man diese berechneten Typen in der Typisierung mitverwenden. Wir betrachten die Situation, in der schon einige Superkombinatoren Typen haben und von anderen verwendet werden, und wollen die Typen der nächsten Schicht berechnen. Die schon berechneten Typen der SK bezeichnen wir mit $\forall(\tau)$, um anzudeuten, dass die Typvariablen in τ umbenannt werden können.
Er startet mit

- vorgegebenen Typ-Annahmen über die Konstruktoren,
- bereits berechneten Typen einiger Superkombinatoren
- den vorgegebenen Typen der Variablen und
- mit $\forall\alpha$ als Typen der Superkombinatoren (deren Typ zu bestimmen ist)

Danach werden die Rümpfe der Superkombinatoren getypt, wobei folgende Regeln verwendet werden. Diese Regeln sind die Regeln für Grundtypen, erweitert um Unifikation. Wir fügen zu den hergeleiteten Typen noch die notwendigen Gleichungen zwischen den Typen hinzu.
Schreibweise: $t : \tau, E$ für das Paar aus hergeleitetem Typ und den Gleichungen E. Eine implizite Annahme ist, dass verschachtelte `cases` keine Variablen mehrfach benutzen, sondern stets neue Variablen benutzen.

Axiom für Variablen
$A \cup x : \tau \vdash x : \tau$ $\quad \tau$ ist in den Annahmen eine Typvariable.

Axiom für Konstanten
$A \cup \{c : \forall\tau\} \vdash \{c : \tau'\}$ $\quad c$ ist ein Konstruktor oder ein bereits getypter Superkombinator. wobei τ' eine umbenannte Version des Typs τ ist mit neuen Typvariablen.

Anwendungsregel

$$\frac{A \vdash a : \tau_1, E_1; A \vdash b : \tau_2, E_2}{A \vdash (a\ b) : \tau_3, E_1 \cup E_2 \cup \{\tau_1 \doteq \tau_2 \to \tau_3\}}$$

wobei τ_3 neue Typ-variable.

Unifikation:

$$\frac{A \vdash a : \tau, E}{A \vdash a : \tau, E'}$$

, wobei E' aus E durch Anwendung von Unifikationsregeln entsteht. wobei kein FAIL auftreten darf.

case-Regel:

$$\begin{array}{l}
A \vdash e : \mu_0, F_0 \\
A \cup \{x_{1,1} : \tau_{1,1} \ldots x_{1,n(1)} : \tau_{1,n(1)}\} \vdash (c_1\ x_{1,1} \ldots x_{1,n(1)}) : \mu_1, F_1 \\
\ldots \\
A \cup \{x_{m,1} : \tau_{1,1} \ldots x_{m,n(m)} : \tau_{1,n(m)}\} \vdash (c_m\ x_{m,1} \ldots x_{m,n(m)}) : \mu_m, F_m \\
A \cup \{x_{1,1} : \tau_{1,1} \ldots x_{1,n(1)} : \tau_{1,n(1)}\} \vdash e_1 : \tau_1, E_1 \\
\ldots \\
A \cup \{x_{m,1} : \tau_{1,1} \ldots x_{m,n(m)} : \tau_{1,n(m)}\} \vdash e_m : \tau_m, E_m \\
\hline
A \vdash (\mathtt{case}_T\ e\ ((c_1\ x_{1,1} \ldots x_{1,n(1)}) \to e_1); \ldots ; ((c_m x_{m,1} \ldots x_{m,n(1)}) \to e_m)) : \tau, \\
F_0 \cup F_1 \cup \ldots \cup F_m \cup \{\mu_0 \doteq \mu_1, \ldots, \mu_0 \doteq \mu_m\} \cup \{\mu_0 \doteq T \ldots\} \cup \\
E_1 \cup \ldots \cup E_m \cup \{\tau \doteq \tau_1, \ldots, \tau \doteq \tau_m\} \\
\tau \text{ ist eine neue Typvariable}
\end{array}$$

Typberechnung:

$$\frac{A \vdash t : \tau, E}{A \vdash t : \sigma(\tau)}$$

wobei σ die Substitution ist, die durch Unifikationsregeln aus E berechenbar ist

Superkombinator-Regel zur Berechnung neuer Annahmen

$$\frac{A \cup \{x_1 : \tau_1, \ldots x_n : \tau_n\} \vdash e : \tau, E}{A \vdash sc : \sigma(\tau_1 \to \ldots \to \tau_n \to \tau)}$$

Wenn σ Lösung von E und wenn `sc x_1 ... x_n = e` Definition für `sc` ist.

Bemerkung 11.2. Die Regel für Abstraktionen ist weggelassen. Es wird angenommen, dass man Abstraktionen zunächst in Superkombinatoren umwandelt und dann typisiert. Bei der Typisierung wird auch angenommen, dass Lamnbdalifting durchgeführt wurde (zumindest implizit).

11.3 Typberechnung mittels Iteration:

Man startet mit Annahmen $A_0 = \{sc_i : \alpha_i \mid i = 1, \ldots, n\}$ und berechnet mit der Superkombinatorregel neue Typen $sc_i : \alpha_i'$. Danach iteriert man

diesen Zyklus, indem man mit den berecheten Typen als neue Annahmen $A' = \{sc_i : \alpha'_i | i = 1, \ldots, n\}$ weitermacht.
Die Iteration terminiert, wenn die Typannahmen konsistent sind.

Es gelten folgende Aussagen zum Typsystem, wobei wir allerdings auf einen Beweis in diesem Skript verzichten:

- Die berechneten Typen für Superkombinatoren nach einem Iterationsschritt sind eindeutig bis auf Umbenennung von Variablen. Deshalb sind auch die berechneten Typen der Superkombinatoren im Falle des Terminierens eindeutig.
- Die erreichten Typannahmen werden in jedem Schritt spezieller, so dass bei dieser Iteration die Abbruchbedingung nur erreicht werden kann, wenn die Typannahmen pro Superkombinator bis auf Umbenennung gleich sind.

Zum Beweis der zweiten Aussage benötigt man eine Monotonie-Eigenschaft: Wenn die Annahmen für die Superkombinatoren spezieller sind, dann sind auch die hergeleiteten Typen spezieller. (ohne Beweis)

Die Überprüfung auf Konsistenz macht man mit einem Subsumptioncheck für polymorphe Typen. τ_1 ist allgemeiner als τ_2, wenn man eine Substitution σ findet mit $\sigma(\tau_1) = \tau_2$. Mit Unifikation ist das leicht zu berechnen, indem man die Variablen in τ_1 umbenennt, dann die Variablen in τ_2 zu Konstanten erklärt und danach unifiziert.

Beispiel 11.3. Typ von `length`:
`length as = case_list as (nil -> 0); ((cons x xs) ->  (1 + length xs)))`
Vorwärtsschließen: Sei $A = \{\texttt{length} : \forall\alpha, \texttt{as} : \tau_1, \texttt{x} : \tau_2, \texttt{xs} : \tau_3, + : \texttt{Int} \rightarrow \texttt{Int} \rightarrow \texttt{Int}\}$
$A \vdash \texttt{length} : \alpha'$
$A \vdash \texttt{(length xs)} : \tau_3 \rightarrow \tau_4 \doteq \alpha'$
$A \vdash \texttt{1+ (length xs)} : \texttt{Int}, \tau_4 \doteq \texttt{Int}$
Verwendung der case-Regel:
Es gilt $0 : \texttt{Int}$ und $\texttt{Nil} : [\beta_1]$
`cons`: $\beta_2 \rightarrow [\beta_2] \rightarrow [\beta_2]$
$A \vdash \texttt{(cons x xs)} : [\beta_2], \{\tau_3 = [\beta_2], \tau_2 = \beta_2\}$
$A \vdash \texttt{(case\_list as nil -> 0; (cons x xs) -> (1 + length xs))} : \tau,$
$\{\tau \doteq \texttt{Int}, \tau_3 \doteq [\beta_2], \tau_2 \doteq \beta_2, \tau_4 \doteq \texttt{Int}, \tau_1 \doteq [\beta_2] \doteq [\beta_1]\}$
Da die meisten Variablen irrelevant sind, genügt es, bis auf die Variablen τ und τ_1, nur die Lösbarkeit der Gleichungen festzustellen. Das Gleichungssystem nach Unifikation sieht so aus: $\{\tau \doteq \texttt{Int}, \tau_1 \doteq \tau_3 \doteq [\tau_2], \tau_4 = \texttt{Int}\}$
Die berechnete Annahme für den Typ von `length` ist dann: `length`: $[\tau_2] \rightarrow \texttt{Int}$.
Danach ist die Annahme für `length`: $\forall.[\tau_2] \rightarrow \texttt{Int}$.
Eine weitere Iteration zeigt die Konsistenz dieser Annahme.

Beispiel 11.4. `g x = (cons 1  (g (g 'c')))`
$\{\texttt{x} : \tau_1, \texttt{g} : \forall a\}$

`(g 'c')` mit $\texttt{g} : \alpha_1$ ergibt zunächst $\alpha_1 = \alpha_2 \to \alpha_3$ und $\alpha_2 = \texttt{Char}$.
`g(g 'c')` ergibt $g : \beta\ \beta \doteq \beta_1 \to \beta_2, \beta_1 \doteq \alpha_3$,
`(cons 1  (g (g 'c')))`: ergibt $\beta_2 = [\texttt{Int}]$
Dies ergibt als nächste Typannahme für `g`: $\forall\alpha.\alpha \to [\texttt{Int}]$. Verwendet man diese, so erhält man, dass dies der richtige Typ ist.

Beispiel 11.5. Betrachte `g x = g 1`:
$\texttt{g} : \alpha \to \beta, x : \tau$ ergibt: `g 1`: β. D.h. `g`: $\alpha \to \beta$.
Dieser Typ bedeutet, dass das Resultat alle Typen hat. Das kann immer so interpretiert werden, dass die Auswertung von `(g x)` zu einer WHNF nicht terminiert. Ein Objekt beliebigen Typs (bzw. eines das alle Typen hat), kann nicht in WHNF sein, denn ein Konstruktor als oberstes Funktionssymbol impliziert, dass der Typ des Objektes mit dem entsprechenden Typkonstruktor anfängt. Ein Funktionsausdruck als WHNF hätte ein $\to$ als obersten Typkonstruktor. Es bleibt nur ein Objekt ohne WHNF, d.h. ein nicht-terminierender Ausdruck. D.h. In gewissen Fällen liefert der Type-checker Nichtterminierungsinformation. Aber nicht jeder nichtterminierende Ausdruck hat einen solchen allgemeinen Typ.

11.4 Frage: Benötigt das Verfahren tatsächlich mehrere Iterationen?

JA:

Betrachte `g x = cons x (g (g 'c'))`

- 1. Iteration
  Annahme $\{\texttt{x} : \tau_1, \texttt{g} : \forall\alpha\}$
  `(g 'c')` : $\alpha_1 \doteq \alpha_2 \to \alpha_3$ und $\alpha_2 \doteq \texttt{Char}$
  `g(g 'c')`
  Annahme: $g : \beta$: $\beta \doteq \beta_1 \to \beta_2, \beta_1 \doteq \alpha_3$,
  `g(g 'c')` : β_2
  `(cons x  (g(g 'c')))`: ergibt $\beta_2 = [\tau_1]$
  Gesamttyp von $\texttt{g} : \tau_1 \to [\tau_1]$
- 2. Iteration
  Neue Annahme: $\{\texttt{x} : \tau_1, \texttt{g} : \forall\alpha \to [\alpha]\}$
  `(g 'c')` : $\alpha_1 \to [\alpha_1]$ und $\alpha_1 = \texttt{Char}$
  `g(g 'c')` ergibt $\texttt{g} : \beta_1 \to [\beta_1], \beta_1 = [\texttt{Char}]$
  `(cons x  (g(g 'c')))`: ergibt $\tau_1 = [\texttt{Char}]$ und als Typ
  `(cons x  (g(g 'c')))`: $[[\texttt{Char}]]$

- 3. Iteration
  Neue Annahme: $\{\texttt{x} : \tau_1, \texttt{g} : [\texttt{Char}] \to [[\texttt{Char}]]\}$
  `(g 'c')`: $[\texttt{Char}] = \texttt{Char}$ liefert FAIL. D.h. der Ausdruck ist nicht typisierbar.

`g` hat also keinen Typ im iterativen Typsystem.

11.5 Frage: Terminiert das Verfahren?

Die Antwort ist NEIN:

Betrachte `g x = cons x (g (g bot))`, wobei *bot* nichtterminierender Superkombinator mit Typ $\forall\gamma$.
Annahme: $\{\mathtt{x} : \tau_1, \mathtt{g} : \forall\alpha, bot : \forall\gamma\}$
`(g bot)`: $\alpha_1 \doteq \alpha_2 \to \alpha_3$ und $\alpha_2 \doteq \gamma_1$
`g(g bot)`: $\mathtt{g} : \beta$ ergibt: $\beta \doteq \beta_1 \to \beta_2, \beta_1 \doteq \alpha_3$,
`(cons x (g(g bot)))`: ergibt $\beta_2 = [\tau_1]$
Der Gesamttyp ist: $\tau_1 \to [\tau_1]$

Neue Annahme: $\{\mathtt{x} : \tau_1, \mathtt{g} : \forall\alpha \to [\alpha], \mathtt{bot} : \forall\gamma\}$
`(g bot)` $:: [\alpha_1]$ und $\alpha_1 \doteq \gamma_1$.
`g(g bot)`: $\mathtt{g} : \alpha_2 \to [\alpha_2], \alpha_2 \doteq [\alpha_1]$,
`(cons x  (g(g bot)))`: ergibt $[[\alpha_1]] \doteq [\tau_1]$
Superkombinatorregel für `g`: $[\alpha_1] \to [[\alpha_1]]$

Wir berechnen dies mit einer allgemeineren Annahme, wobei wir die n-fache Anwendung des List-Konstruktors mit $[\cdot]^n$ notieren.
$\{x : \tau_1, \mathtt{g} : \forall[\alpha]^n \to [\alpha]^{n+1}, \mathtt{bot} : \gamma\}$

`(g bot)`	Hat Typ $[\alpha_1]^{n+1}$
`g(g bot)`	$\mathtt{g} : [\alpha_2]^n \to [\alpha_2]^{n+1}$, $[\alpha_2]^n \doteq [\alpha_1]^{n+1} \Rightarrow \alpha_2 \doteq [\alpha_1]$. Es ergibt sich: `g(g bot)` $: [\alpha_1]^{n+2}$
`cons x (g (g bot))`	Wegen $\mathtt{cons} : \beta \to [\beta] \to [\beta]$ ergibt sich $[\tau_1] \doteq [\alpha_1]^{n+2}$ und somit $\tau_1 \doteq [\alpha_1]^{n+1}$. Der Resultattyp ist `cons x (g (g bot))` $: [\alpha_1]^{n+2}$

Der hergeleitete Typ von `g` ist dann: $[\alpha_1]^{n+1} \to [\alpha_1]^{n+2}$

Da der Typ immer tiefer geschachtelt wird, terminiert dieses Verfahren nicht.

Beispiel 11.6. : Ein weiteres Beispiel zur Nichtterminierung, das leichter zu überschauen ist, aber zwei rekursive Superkombinatoren benötigt:
Man kann konstante Superkombinatoren A, B definieren:

```
A = [B]
B = [A]
```

Man wird mit dem hier angegebenen Verfahren iterativ immer tiefer geschachtelte Typen bekommen. Denn die Annahme $\mathtt{A} : \tau_1, \mathtt{B} : \tau_2$ mit beliebigen Typtermen τ_1 und τ_2 liefert danach $\mathtt{A} : [\tau_2], \mathtt{B} : [\tau_1]$. Typkonsistenz kann nicht erfüllt sein, da wir dann eine zyklische Abhängigkeit hätten: $\tau_1 \subseteq [\tau_2]$, und $\tau_2 \subseteq [\tau_1]$ dies kann nur sein, wenn $\tau_1 \subseteq [[\tau_1]]$, was niemals stimmt. Damit kann das iterative Typinferenzverfahren nicht terminieren.

Satz 11.7. *Es gilt: Typisierung (mit konsistenten Typannahmen) ist unentscheidbar.*

Das Kern des Problems ist die sogenannten Semi-Unifikation, die unentscheidbar ist.

11.6 Terminierung:

Die Terminierung kann auf Kosten der Berechnung des allgemeinsten Typs erzwungen werden (bzw. auf die Gefahr hin, dass ein Ausdruck als ungetypt abgelehnt wird), wenn man nach einigen Iterationen einen "Milner"-Iterationsschritt anschließt:

> Erlaube nur noch Instanzen des Typs in der Typ-Annahme ohne Umbenennung der Typvariablen. Anschließend unifiziere den berechneten Typ eines Superkombinators mit dem Typ in der Annahme.

Danach ist dann nämlich die Bedingung der konsistenten Typannahme automatisch erfüllt.

Beispiel 11.8. `g x = cons x (g (g bot))`
Das iterative Typisierungsverfahren terminiert nicht für dieses Beispiel. Das Verbot von Kopien des Typs des aktuell zu typisierenden Superkombinators ergibt:
$\{\mathtt{x} : \tau_1, \mathtt{g} : \forall\alpha \to [\alpha], \mathtt{bot} : \gamma\}$
$(\mathtt{g\ bot})$: $\alpha \to [\alpha]$ und $\alpha \doteq \gamma_1$
$\mathtt{g(g\ c)}$ muss getypt sein $\Rightarrow \mathtt{g} : \alpha \to [\alpha], \alpha \doteq [\alpha]$:
Die letzte Gleichung $\alpha \doteq [\alpha]$ ergibt einen Fehler bei der Unifikation.

11.7 Typisierungsverfahren von Milner

Das Typisierungsverfahren von Milner geht genauso vor wie das iterative Verfahren, mit zwei Unterschieden,

- Superkombinatoren werden unterteilt in bereits typisierte und noch zu typisierende
- Von zu typisierenden Superkombinatoren wird nur eine Kopie des Typs erlaubt.

Praktisch wird es immer folgendermaßen verwendet: Man macht eine Abhängigkeitsanalyse anhand der Aufrufhierarchie und der bereits typisierten Superkombinatoren. Die Typen der Funktionen werden von unten her berechnet. Diese Vorgehensweise berechnet i.a. allgemeinere Typen als "alle auf einmal", da für die bereits typisierten SK Kopien der Typen möglich sind.
Dieses Verfahren liefert eingeschränktere Typen als das oben beschriebene iterative Verfahren, oder lehnt womöglich den (iterativ wohl-getypten) Ausdruck als ungetypt ab. Das Verfahren hat den Vorzug, dass es terminiert und eindeutige Typen liefert. Es wird deshalb in mehreren Programmiersprachen verwendet. Da Programmiersprachen wie Haskell, Miranda, ML, ... letrec- und lambda-Konstrukte haben, ist das Typisierungsverfahren etwas komplizierter als das hier beschriebene, denn es muß lokal rekursiv definierte Funktionen ebenfalls typen können.
Die worst-case Komplexität des Problems "ist eine Funktion Milner-typisierbar?" ist EXPSPACE-hard. Dies wirkt sich so aus, dass es Funktionen gibt, deren Typ im Milner Typsystem exponentiell viele Typvariablen benötigt.

Das Milner Typsystem macht Unterschiede zwischen let-gebundenen und lambda-gebundenen Variablen: let-gebundene dürfen polymorph sein, lambda-gebundene Variablen müssen monomorph sein. Außerdem dürfen rekursiv definierte Superkombinatoren im rekursiven Aufruf nur mit einem Typ benutzt werden.
Wir wollen die guten Eigenschaften des Milner-Typsystems begründen:

Aussage 11.9. *Werden für die zu typiserenden Superkombinatoren keine umbenannten Kopien von Typen erlaubt, dann terminiert das (Milner-)Typisierungsverfahren im ersten Iterationsschritt.*

Begründung. Das Verfahren startet mit der Annahme $A = A_0 \cup \{sc_i : \alpha_i\}$, wobei A_0 die Konstruktoren und bereits typisierten Superkombinatoren enthält und sc_i die zu typisierenden Superkombinatoren sind. Da keine Kopien des Typs von Superkombinatoren sc_i gemacht werden, können alle Anforderungen an den Typ der Superkombinatoren sc_i berechnet und danach unifiziert werden. Am Ende ergibt sich entweder ein FAIL beim Unifizieren, oder die Unifikation ist erfolgreich und der für sc_i angenommene Typ ist gleich dem berechneten Typ.

Beispiel 11.10. Milner-Methode: Berechne den Typ der Funktion `map`.

```
map f as = case_list as  (nil -> nil)
                         ((x:xs) ->  (f x) : (map f xs))
```

Die Typannahme ist $A = \{\texttt{map} : \alpha, \texttt{f} : \tau_1, \texttt{as} : \tau_2, \texttt{x} : \tau_3, \texttt{xs} : \tau_4\}$.
Typ-inferenz für `map` ergibt:

1. `map f xs` getypt $\Rightarrow$ $\texttt{map} : \alpha_1 \to \alpha_2 \to \alpha_3$, $\alpha_1 \to \alpha_2 \to \alpha_3 \doteq \alpha$, $\alpha_1 \doteq \tau_1, \alpha_2 \doteq \tau_4$.
2. `(f x)` getypt $\Rightarrow \tau_1 \doteq \tau_3 \to \tau_5$
3. `(f x) : map f xs` getypt: `cons`: $\tau_6 \to [\tau_6] \to [\tau_6]$, $\tau_5 \doteq \tau_6, [\tau_6] \doteq \alpha_3$
4. `Nil` und `(f x) : map f xs` haben den gleichen Typ: $[\beta_1] \doteq [\tau_6]$
5. `x:xs` ist getypt: $\tau_4 \doteq [\tau_3]$
6. `as, nil` und `x:xs` haben gleichen Typ: $\tau_2 \doteq [\beta_2] \doteq [\tau_3]$
7. Ergebnis ist vom Typ $[\tau_6] \doteq \alpha_3$
8. Unifikation ergibt: $\tau_5 \doteq \tau_6 \doteq \beta_1, \tau_3 \doteq \beta_2, \tau_4 \doteq \tau_2 \doteq [\tau_3]$ $\alpha_1 \doteq \tau_1 \doteq \tau_3 \to \tau_5, \alpha_2 \doteq \tau_4, \alpha_3 \doteq \tau_2 \doteq [\tau_5]$

Insgesamt ergibt sich: `map`: $(\tau_3 \to \tau_5) \to [\tau_3] \to [\tau_5]$ oder, standardisiert:
`map` : $(\alpha \to \beta) \to [\alpha] \to [\beta]$

Beispiel 11.11. Der Milner Algorithmus kann Typfehler liefern, obwohl der iterative Algorithmus einen Typ liefert: z.B. für die Funktion `g x = 1: g(g 'c')` liefert der Milner-Algorithmus, dass diese nicht wohlgetypt ist, obwohl wir iterativ dafür den Typ: $\alpha \to$ `[Int]` berechnet haben:
`g`: $\alpha_1 \to \alpha_2, \texttt{x} : \alpha_1$,
`g 'c'`: $\alpha_2, \alpha_1 =$ `Char`
`g (g 'c')`: $\alpha_2; \alpha_2 =$ `Char`
`1:g(g 'c')` erfordert mit `cons`: $\beta \to [\beta] \to [\beta]$

$\beta \doteq$ `Int`, [`Int`] = [`Char`]
Das ist nicht unifizierbar.
Ein Test in Gofer liefert: nicht (Gofer-)typisierbar.

Bemerkung 11.12. : Zur Typisierung in Haskell: In Haskell und Gofer ist der Superkombinator `g x = cons x (g (g 'c'))` nicht typisierbar. Der Grund ist das Typsystem von Haskell / Gofer, das auch Typklassen unterstützt und Überladung auflöst. Die Überladungsauflösung soll zur Compilezeit auf einfache Weise möglich sein. Dies wird unterstützt durch die Berechnung von eingeschränkteren Typen (nach Milner).

Bemerkung 11.13. Das Verfahren verhält sich besser, wenn man die Typen der Superkombinatoren nicht berechnen läßt, sondern vorgibt (bzw. vom Programmierer vorgeben läßt), und dann vom System nachrechnen läßt.

Definition 11.14. *Ein Typ τ_1 ist allgemeiner als ein Typ τ_2, wenn es eine Substitution σ für die Typvariablen gibt, so dass $\sigma(\tau_1) = \tau_2$. Dieser Test ist in linearer Zeit durchführbar durch strukturelles Durchmustern der beiden Typen.*

Erhält man mit dem Milner Typberechnungsverfahren einen allgemeineren Typ für einen Superkombinator, so ist das nicht weiter schlimm, aber die Information könnte nützlich für den Programmierer sein. Erhält man einen spezielleren oder einen anderen Typ als den vorgegebenen, so ist die vorgegebene Typisierung vom Typ-checker abzulehnen. Spezieller darf der Typ nicht sein, da dann die Konsistenz der Annahmen verletzt wird.

Übungsaufgabe 11.15.

1. Berechne den Typ des Superkombinators `Fix` mit Definition

   ```
   FIX x = x (FIX x)
   ```

2. Berechne den Typ von

   ```
   \f.(\ x. f (x x)) (\x. f (x x)),
   ```

   wobei diese Definition zuerst in KFP zu transformieren ist:

   ```
   FIX-H f x = f (x x)
   FIX2 f = (FIX-H f) (FIX-H f)
   ```

3. Berechne den Typ von `S` definiert als

   ```
   S x y z = x y (x z)
   ```

4. `ff xs@(y:ys) = length xs + (ff y) + (ff  ys)` Typisiere nach den zwei verschiedenen Methoden

11.8 Bemerkung zur Erweiterung von Typsystemen

Es ist möglich, ein Typsystem zu definieren, das es erlaubt, Bäume mit geschachtelten Listen zu implementieren. Dafür muß man aber bei der Unifikation den Occurs-check weglassen, und muß rekursiv definierte Typen akzeptieren, die sich nicht als endliche Terme darstellen lassen. Ein weiterer Nachteil ist, dass eine bestimmte Klasse von Typfehlern manchmal unentdeckt bleibt. Es sind gerade solche Typfehler, die eine "falsche" Schachtelung von Datenstrukturen charakterisieren. Fehler in Haskell: "... infinite type ... ".

Beispiel 11.16. Argument-ignorierende Funktion: `f  x = f`
$A : \{\texttt{f} : \tau_1 ; \texttt{x} : \tau_2\}$ Typ von `f`: $\tau_2 \to \tau_1$ Unifikation mit τ_1 ergibt $\tau_1 \doteq \tau_2 \to \tau_1$. Standardunifikation liefert einen Fehler. Weglassen des Occurs-check würde diesen Typ zulassen.

Beispiel 11.17. Funktionen auf geschachtelten Listen:

Die Funktion `nn` soll Anzahl der `Nil`-Vorkommen in einer geschachtelten Liste zählen: `nn xs = (case_list xs (nil -> 1) ((cons y ys) -> (+ (nn y) (nn ys))))`
iterativer Typcheck:
$A = \{\texttt{nn} : [\alpha] \to \texttt{Int}, xs : \tau\}$
Der (`case`-Ausdruck hat Typ `Int`.
`xs`: $[\alpha_1] \doteq \tau$
$\texttt{y} : \alpha_1, \texttt{ys} : [\alpha_1], (\texttt{nn y}) : [\alpha_3] \doteq \alpha_1$
ergibt den neuen Typ: $[[\alpha]] \to \texttt{Int}$. Man sieht: Die Iteration terminiert nicht.
In Haskell ist diese Funktion nicht typisierbar und wird zurückgewiesen:

```
nn nil = 1
nn (x:xs) = (nn x) + (nn xs)
```

Der Typ von `nn` darf während des Typcheckens nicht kopiert werden. Das ergibt:
$\{\texttt{nn} : [\alpha] \to \texttt{Int}, \texttt{xs} : \tau\}$
Der `case`-Ausdruck hat Typ `Int`.
`xs`: $[\alpha_1] \doteq \tau$
$\texttt{y} : \alpha_1, \texttt{ys} : [\alpha_1]$ `(nn x)`: $[\alpha] \doteq \alpha_1$ `(nn xs)`: $[\alpha] \doteq [\alpha_1]$
Unifikation (ohne Occurs-check) ergibt hier $\alpha \doteq \alpha_1 \doteq [\alpha]$.
Erlaubt man Unifikation ohne Occurs-check, so könnte man als Typ folgenden Ausdruck nehmen:

$$nn :: [\alpha] \to \texttt{Int} \text{ where } \alpha = [\alpha]$$

Da `Nil` : $[\alpha]$, wobei α beliebigen Typ haben kann, wird in diesem Typsystem eine verschachtelte Liste beschrieben, wobei nur `Nil` oder bot:α als Blatt erlaubt ist. Man kann auf dieser Idee ein Typsystem aufbauen, das ebenfalls dynamische Typfehler verhindert. Allerdings erhält man andere Nachteile, wie z.B. nicht erkannte Fehler beim Programmieren.
Empfehlung: Man sollte die Typcheckvariante mit Occurs-check benutzen.

12 Änderung der Auswertungsreihenfolge: strikt

Die gezielte Änderung der Auswertungsreihenfolge ist eine Optimierung, die von allen Kompilern für nicht-strikte FPS verwendet wird. Auch hier ist wesentlich, dass die Semantik, sprich die kontextuelle Äquivalenz erhalten bleibt.

Beispiel 12.1. Betrachte nochmal die Funktionen

```
NT x    =  NT x
K  x y  =  x
```

Ändert man die Auswertungsreihenfolge so ab, dass die Funktion `K` zunächst das erste, dann das zweite Argument auswertet und dann das Resultat berechnet, dann terminiert die Auswertung von `K` 1 *bot* nicht mehr, obwohl (`K` 1 *bot*)$\Downarrow$. D.h. offenbar ist die Semantik verändert.

Ändert man die Auswertungsreihenfolge so ab, dass die Funktion `K` zunächst das erste Argument auswertet und dann das Resultat berechnet, dann ist der Fehler behoben. Eine erste Vermutung ist, dass diese Änderung richtig ist, aber der Nachweis fehlt.

Im folgenden werden wir den Nachweis erbringen, dass diese Veränderung die Semantik erhält.
Zuerst brauchen wir noch eine Betrachtung zur Normalordnungsreduktion.

Lemma 12.2. *Wenn ein geschlossener Ausdruck eine Normalordnungsreduktion $R[s] \rightarrow_{no} R[s_1]$ hat, wobei R ein Reduktionskontext ist, dann wird die Normalordnungsreduktion den Unterausdruck $R[s_1]$ zunächst solange reduzieren, bis er selbst in WHNF ist. D.h. die Normalordnung hat zwei Möglichkeiten:*

- $R[s] \rightarrow_{no} R[s_1] \rightarrow_{no} R[s_2] \rightarrow_{no} \ldots \rightarrow_{no} R[s_n]$ *s_n ist in WHNF, und danach wird evtl. auch R durch die Normalordnungsreduktion verändert*
- $R[s] \rightarrow_{no} R[s_1] \rightarrow_{no} R[s_2] \rightarrow_{no} \ldots$ *und der Ausdruck s_2 wird niemals zur WHNF reduziert.*

Zunächst definieren wir Striktheit.

Definition 12.3. *Ein KFP-Ausdruck f ist* strikt, *gdw. $(f\ bot)\Uparrow$. Hierbei ist bot ein nichtterminierender Ausdruck.*

Eine Erweiterung ist die Definition der Striktheit in einem Argument:

Definition 12.4. *Die (geschlossene) Funktion f ist strikt im i-ten Argument, gdw. für alle geschlossenen Ausdrücke t_j: $(f\ t_1 \ldots t_{i-1}\ bot)\Uparrow$.*

Man könnte auch eine (i, n)-Striktheit definieren: gdw. für alle geschlossenen Ausdrücke t_j: $(f\ t_1 \ldots t_{i-1}\ bott_{i+1} \ldots t_n)\Uparrow$.

Definition 12.5. *Sei f strikt und s ein Ausdruck. Dann sei die f-*strikte Auswertung *diejenige, die bei Auswertung von $f\ s$ zuerst s mit f-strikter Auswertung in WHNF überführt, und ansonsten wie die Normalordnung reduziert.*

Lemma 12.6. *Wenn f strikt ist, und $(f\ s){\Downarrow}$, dann gilt $s{\Downarrow}$, und die Anzahl der Normalordnungsreduktionen von s ist echt kleiner als die Anzahl der Normalordnungsreduktionen von $f\ s$*

Beweis. Es ist klar, dass $s{\Downarrow}$. Was noch fehlt, ist die Aussage zur Anzahl. Markiert man den Term s und alle während der Reduktion entstehenden Kopien, dann gibt es zwei Fälle:

1. Die Normalordnungs-Reduktion von $f\ s$ reduziert einen der markierten Unterterme s während der Reduktion ebenfalls als Subprozedur zur WHNF. Dann gilt die Behauptung.
2. Die Normalordnungs-Reduktion reduziert niemals die markierten Kopien von s. Dann kann man aber s durch *bot* ersetzen und die Normalordnungs-Reduktion terminiert immer noch. Das ist ein Widerspruch zur Annahme der Striktheit von f. □

Aussage 12.7. *Sei f strikt. Dann $s{\Downarrow}$ gdw. die f-strikte Auswertung von s mit einer WHNF terminiert.*

Beweis. Wenn die f-strikte Auswertung von s mit einer WHNF terminiert, dann gilt $s{\Downarrow}$ wegen Satz 5.11.
Sei die Umkehrung falsch. Dann können wir einen Ausdruck s_0 finden, so dass die Normalordnung terminiert, aber nicht die f-strikte Auswertung. Wir wählen die Anzahl der Normalordnungsreduktionen minimal und als zweiten Parameter die Größe von s_0 minimal.
Falls der Normalordnungsredex nicht von der Form $f\ s$ ist, stimmen die Reduktionen überein. Damit ist das Beispiel nicht minimal. Also ist der Normalordnungsredex von der Form $f\ s$ und $s_0 \equiv R[(f\ s)]$. Da f strikt ist, gilt $s{\Downarrow}$. Nach Lemma 12.6 ist die Anzahl der Normalordnungsreduktionen von s kleiner als die von $f\ s$. Wegen der Minimalität terminiert die f-strikte Reduktion von s.
Jetzt betrachten wir die f-strikte Reduktionsfolge $R[f\ s] \to R[f\ s_1] \to \ldots \to R[f\ s_n] \xrightarrow{no} R[r[s_n/x]]$ wobei $f = \lambda x\ .\ r$. Es gilt $R[f\ s] \xrightarrow{no} R[r[s/x]]$.
Aus Aussage 5.5 und der Tatsache $R[r[s/x]] \xrightarrow{*} R[r[s_n/x]$ können wir jetzt schließen, dass die Anzahl der Normalordnungsreduktionen von $R[r[s_n/x]]$ kleiner als die von $R[f\ s]$ ist, also gilt für diesen Ausdruck die Behauptung. Damit gilt die Behauptung auch für den Ausdruck $R[f\ s]$, da wir eine terminierende f-strikte Reduktion durch Zusammensetzen konstruiert haben.

□

Das gilt auch, wenn wir statt eines einzigen Superkombinators f eine Menge von Funktionen nehmen.

Definition 12.8. *Die Sprache KFP erweitern wir um den Operator* `strict`*, der an Funktionspositionen vorkommen darf. Die Sprache nennen wir KFP+strikt. Für einen Reduktionskontext R wird der Ausdruck $R[((\texttt{strict}\ t)\ s)]$ in*

KFP+strikt-Normalordnung so reduziert, dass zuerst s zu (in KFP+strikt-Normalordnung) WHNF s' reduziert, und dann $R[(t\ s')]$ in Normalordnung weiter reduziert wird.

Die Verwendung des Operators `strict` in KFP nur für strikte Funktionen bewirkt keine Änderung der kontextuellen Äquivalenz.
Aber: die Einführung des Operators `strict` zur Veränderung der Normalordnung in allgemeiner Form birgt noch eine Unklarheit: Es könnte sein, dass sich dadurch die kontextuelle Gleichheit von Ausdrücken verändert, da man danach mehr Testprogramme hat, die Ausdrücke auf Terminierung prüfen. D.h. die Erweiterung von KFP zu KFP+strikt könnte nicht konservativ sein. Das ist tatsächlich so :

Aussage 12.9. *Die Einführung des Operators* `strict` *verändert die kontextuelle Gleichheit von Programmen beim Übergang von KFP nach KFP+strikt.*

Beweis. Wir betrachten den Kontext $D[] = \lambda f\ .\ case_{\mathtt{Bool}}\ (f\ [])\ (\mathtt{True} \rightarrow \mathtt{True})\ (\mathtt{True} \rightarrow \mathtt{False})$ und die beiden Ausdrücke
$s_0 = D[\lambda x\ .\ \bot)]$ und $t_0 = D[\bot)]$.
Behauptung: In KFP sind diese Ausdrücke nicht unterscheidbar. D.h. $s_0 \sim_c t_0$.
Sei dazu C ein beliebiger Kontext. Wir zeigen, dass beide Ausdrücke dasselbe Resultat bzgl Terminierung ergeben. Wir machen eine Reduktion von $C[s_0]$ und $C[s_0]$ mit Markierungen, wobei sowohl der case-Ausdruck in D markiert ist als auch der Ausdruck $\lambda x\ .\ bot$ und bot.
Es genügt die Betrachtung, dass die Resultate der Normalordnungsreduktion von $s_0\ r$ und $t_0\ r$ immer gleich sind. Es wird zunächst $r\ (\lambda x\ .\ bot)$ reduziert, parallel dazu $r\ (bot)$. Falls die Normalordnungsreduktion den markierten Term $(\lambda x\ .\ bot)$ oder (bot) verwendet, ist das Ergebnis Nichtterminierung, also äquivalent zu bot. Wenn nicht, dann ist das Ergebnis auf beiden Seiten das gleiche. Da danach ein $\mathtt{case}_{\mathtt{Bool}}$ auf das Ergebnis gemacht wird, ist das entweder ebenfalls bot, oder falls es einer der Werte `True` oder `False` ist, das Ergebnis `True`. Hier kommt es darauf an, dass das $\lambda x\ .\ bot$ nicht von einem Konstruktor verpackt werden kann. Im Kontext C wird somit ebenfalls immer das gleiche Resultat erhalten.
D.h. die Ausdrücke s_0, t_0 sind in KFP nicht unterscheidbar.
Behauptung: s_0, t_0 sind unterscheidbar, wenn es den Operator `strict` gibt; d.h. in KFP+strikt. Dazu wende s_0, t_0 jeweils an auf: $\mathtt{strict}(\lambda x\ .\ \mathtt{True})$. Das ergibt für s_0: da $\lambda x\ .\ bot$ in WHNF, ergibt sich `True` als Resultat. Für t_0 ergibt sich bot als Resultat.

□

Wenn man sich die Verwendung von `strict` genauer anschaut, so sind das meistens Verwendungen, die entweder strikte Funktionen betreffen, oder Argumente, die in SCWHNF erwartet werden. Diese Art der Verwendung ist konservativ, wenn sie nur strikte Argumente von Funktionen betrifft, oder wenn die strikt gemachte Funktion auch implementiert werden könnte.

Bemerkung 12.10. Für die Reduktion in der Sprache KFP+strikt kann man die gleichen Überlegungen anstellen wie für KFP und erhält dann angepasste Begriffe (z.B. Reduktionskontext). Eine neue Reduktion ist:

$$\text{strict} \quad \frac{((\texttt{strict}\ f)\ r)}{(f\ r)} \quad \text{Wenn } r \text{ eine WHNF ist}$$

Für die Regeln der Kernsprache KFP+strikt gilt ebenfalls, dass sie eine korrekte Programmtransformation ist.
Somit modelliert die Kernsprache KFP+strikt die Auswertung sowohl für strikte als auch nicht-strikte FPS mit case und Konstruktoren.
Aber auch hier ist wieder eine Warnung angebracht: In einer strikten FPS sind alle Abstraktionen und alle Konstruktoren strikt, nur das case-Konstrukt nicht, so dass man als Bild aller Ausdrücke nur eine Teilmenge von KFP+strikt hat, was wieder eine veränderte kontextuelle Ordnung bewirkt: Z.B. ist $\lambda f\ .\ bot =_c \lambda f\ .\ f\ bot$ in einer strikten FPS, da man nur strikte Objekte für f einsetzen kann. Diese Funktionen sind in KFP aber unterscheidbar, indem man sie auf $\lambda x\ .\ \texttt{True}$ anwendet.
Es sieht so aus, also ob diese Unterschiede sich aufheben, wenn man einfach $bot \sim_c \lambda x\ .\ bot$ setzt und dann $\sim_c$ entsprechend anpasst. Das kann allerdings nur funktionieren, wenn der `strict`-Operator angepasst wird, so dass er eine Berechnung (Auswertung) erzwingt, die erst stoppt, wenn garantiert ist, dass die ausgewerteten Objekte nicht äquivalente zu $\perp$ sind. Das ist unrealistisch, da es keine einfache Form dieser Berechnung gibt.

Bemerkung 12.11. Eine weitere charakteristische Beziehung, ist

$$\lambda x.bot \leq_{c,KFP} c\ t_1 \ldots t_{ar(n)}$$

Ist der Konstruktor `c` strikt, dann wird diese Beziehung falsch.
Wenn der Konstruktor nicht-strikt ist, dann kann man sich überlegen, dass immer noch $\lambda x.bot \leq_{c,KFP+strict} c\ t_1 \ldots t_{ar(n)}$ gilt.
Fügt man einen Operator `hypereval` hinzu, der Ausdrücke in der Tiefe auswertet:

- `hypereval`$(\lambda x.r) \to (\lambda x.r)$ (Auswertung beendet) item `hypereval`$(t) \to$ `hypereval`(t'), falls $t \xrightarrow{no} t'$ (Auswertung)
- `hypereval`$(c\ t_1 \ldots\ t_{ar(c)}) \to c$ (`hypereval` t_1)...(`hypereval` $t_{ar(c)}$)

Danach gilt die Beziehung nicht mehr, denn $\lambda x.bot$ wird nicht weiter ausgewertet, während z.B. `repeat 1` nicht terminiert unter der Auswertung von `hypereval`.

Als **Fazit** können wir festhalten, dass

- nach dem Hinzufügen des Operators `strict` (mit freier Verwendung) zu Haskell offenbar die Kernsprache KFP nicht als Basis dienen kann, sondern KFP+strikt.

- KFP+strikt auch nicht die kontextuelle Gleichheit der strikten FPS exakt abbildet.
- Gute Nachricht: Alle Gleichheiten, die in KFP+strikt gelten, gelten sowohl in Haskell, als auch in strikten FPS. Nur die Ungleichheiten, kann man nicht alle übernehmen, oder anders ausgedrückt: nicht alle in Haskell bzw. strikten FPS gleichen Ausdrücke sind auch KFP+strikt gleich.
- KFP als exakte Basis ist verwendbar, wenn nur Argumente von strikten Funktionen mit `strict` markiert werden können.
- Vermutlich ist die kontextuelle Gleichheit die durch Hinzufügen von $bot = \lambda x.bot$ entsteht, für (getypte) strikte FPS verwendbar, wenn diese Argumente von Funktionstyp lazy behandeln, d.h. nicht auswerten.

12.1 Strikte Superkombinatoren

Definition 12.12. *Eine Funktion (Superkombinator) f mit Definition $f\ x_1\ \dots\ x_n = t$ heißt* strikt *in k-ten Argument, wenn folgendes gilt:*

> *wenn $a_k, 1 \leq k \leq n$ keine WHNF hat, dann gilt für alle Ausdrücke $a_1, \dots, a_n$: $(f\ a_1 \dots a_n)$ hat keine WHNF.*

Da man als semantischen Wert alle Ausdrücke ohne WHNF mit $\perp$ zusammenfaßt, schreibt man dies auch oft kürzer als

$$(f\ a_1 \dots a_{k-1}\ \perp\ a_{k+1} \dots a_n) = \perp$$

Eine Funktion f mit Definition $f\ x_1\ \dots\ x_n = t$ heißt hyper-strikt *in k-ten Argument, wenn für alle Ausdrücke $a_1, \dots, a_n$: wenn $a_k, 1 \leq k \leq n$ keine Normalform hat, dann hat $(f\ a_1\ \dots\ a_n)$ keine NF.*

Die Striktheit von f kann zwei verschiedene Ursachen haben:

- Alle Ausdrücke der Form $(f\ a_1\ \dots\ a_n)$ haben keine WHNF (f terminiert nicht), oder
- Jede Auswertung von $(f\ a_1\ \dots\ a_n)$ zur WHNF muß irgendwann auch a_k zur WHNF auswerten.

Es gilt: Striktheit ist eine unentscheidbare Eigenschaft von Funktionen, da diese das Halteproblem für Turingmaschinen umfaßt.
Das Wissen über Striktheit von Funktionen kann in der Auswertungsstrategie ausgenutzt werden.
Hat man mehrere strikte Argumentpositionen von Funktionen, so legt man im allgemeinen noch eine Reihenfolge von links nach rechts fest, aber auch eine parallele Auswertung zur WHNF könnte angestoßen werden.

Lemma 12.13. *Der n-stellige Superkombinator f sei strikt im k-ten Argument. Wenn $(f\ a_1\ \dots\ a_n)\Downarrow$, dann gilt auch $a_k\Downarrow$.*

Beweis. Gibt es $a_1, \dots, a_{k-1}, a_{k+1}, \dots, a_n$, so daß a_k nicht zu WHNF reduziert wird, dann kann man statt a_k auch bot einsetzen. Die Reduktion ändert sich dadurch nicht. Dies ist aber ein Widerspruch zur Striktheit von f im k-ten Argument. $\square$

13 Verzögerte (Lazy) Auswertung

Die Auswertungsidee bei nicht-strikten funktionalen Programmier-Sprachen kann man folgendermaßen beschreiben: Es gibt einen Auftrag: "werte aus (zu WHNF, oder zu Normalform)". Dieser Auftrag kann Unteraufträge anstoßen. Der oberste Auftrag wird auch dann erfüllt, wenn immer nur die unbedingt erforderlichen Unteraufträge gestartet werden, d.h wenn nur das allernotwendigste ausgewertet wird. Die Normalordnungsreduktion entspricht dieser Auswertungsstrategie, wenn zu WHNF ausgewertet werden soll.
Der Auftrag "werte aus Normalform aus" kann in ähnlicher Weise erfolgen. Allerdings gibt es verschiedene Möglichkeiten:

1. Die Strategie, den Auftrag "werte zu Normalform aus", ungeprüft nach unten weiterzugeben hat manchmal den Nachteil, daß zu viel ausgewertet wird.
2. Die strikte Auswertung liefert zwar immer eine Normalform, wenn sie terminiert, aber sie kann den Auftrag nicht immer erfüllen, dann manchmal terminiert diese nicht, obwohl eine Normalform existiert.
3. Ein Strategie, die die Normalform, falls sie existiert, immer findet ist:
   (a) Werte zu WHNF aus.
   (b) Werte die Komponenten der WHNF ebenfalls zu WHNF aus, usw rekursiv.

Erinnerung: Ein Ausdruck ist in WHNF, wenn er von folgender Form ist:

1. $(c\ t_1 \ldots t_n)$ und c ist ein Konstruktor
2. $(f\ t_1 \ldots t_n)$ und f ist ein Superkombinator.
3. $\lambda x.s$

Wir vergleichen den Ressourcenbedarf von Implementierungen der verschiedenen Reihenfolgen der Auswertung:
Vergleicht man die Reduktion in Anwendungsordnung und Normalordnung auf Ausdrücken in Baumform, dann stellt man fest: Beim Reduzieren der Ausdrücke (wie bisher) kann die Reduktion in Normalordnung exponentiell mehr Platz und Zeit benötigen als die Reduktion in Anwendungsordnung:

Beispiel 13.1. Wir berechnen 2^n auf eine etwas merkwürdige Weise:

```
dd n x = (if n == 0 then x else (dd (n - 1) (x + x)))
```

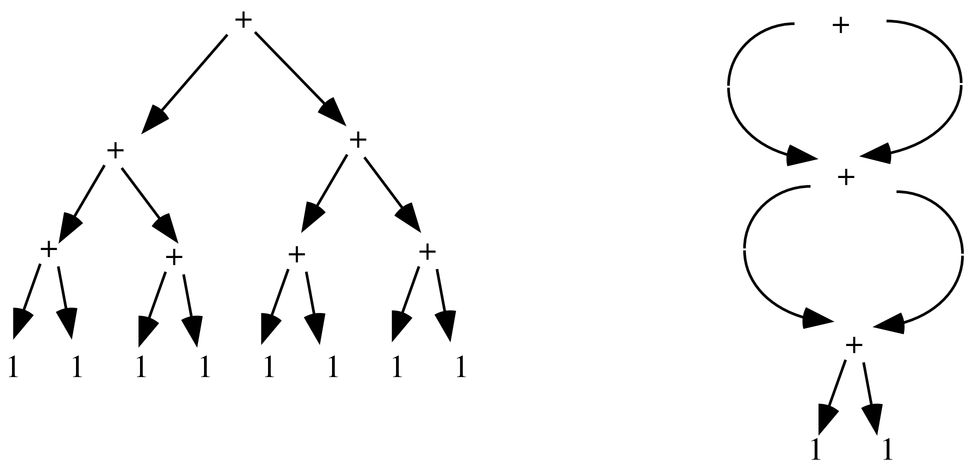
`(dd n 1)` ergibt dann 2^n.
Die Auswertung in Anwendungsordnung sieht so aus:

```
(dd 10 1) → (if 10 == 0 then 1 else (dd (10 - 1) (1 + 1)))
          → (dd (10 - 1) (1 + 1))
          → (dd 9 2)
          → (if  9 == 0 then 2 else (dd (9 - 1) (2 + 2)))
          → (dd 8 4)
          → (if  8 == 0 then 4 else (dd (8 - 1) (4 + 4)))
          ...
          ...
          → 1024
```

Die Auswertung in Normalordnung sieht so aus:

```
(dd 10 1) → (if 10 == 0 then 1 else  (dd (10 - 1) (1 + 1)))
          →  (dd (10 - 1) (1 + 1))
          →  (if  10 - 1 == 0  then (1 + 1)
               else (dd ((10 - 1) - 1) ((1 + 1) + (1 + 1))))
.....................................................................
```

Man sieht, daß zunächst ein +-Ausdruck mit 2^n 1'en aufgebaut wird, der dann erst addiert wird.
!!! Dies ist aber ein selbst geschaffenes Problem, denn die Exponentialität kam durch das explizite Kopieren z.B. des Ausdrucks $(1 + 1)$.
Lösung: Man ersetzt den Baum durch einen gerichteten Graphen. D.h. man verwendet und verwaltet gemeinsame Knoten (node-sharing) bei der Reduktion. Wir werden sehen, dass danach die Anzahl der Reduktionsschritte in dieser Auswertung bei Anwendungsordnung und Normalordnung wieder gleich ist .



13.1 Anzahl der Reduktionen in DAG-Darstellung

Wir simulieren die gemeinsamen Knoten durch einen Baum mit Gleichheitsmarkierungen (z.B. eine Zahl). Bei der Beta-Reduktion werden Argumente in der Simulation kopiert, aber mit gleicher Markierung versehen. Die Beta-Reduktion wird jeweils analog auf allen gleich markierten Termen durchgeführt (und zählt dann nur als ein Schritt). Ein Term der unmittelbar reduziert wird, verliert seine Markierung auf der obersten Ebene.

Definition 13.2. *Die Reduktionsstrategie, die Normalordnung verwendet und gemeinsame Knoten verwendet (shared nodes), nennt man* verzögerte Reduktion (lazy reduction). *Wir bezeichnen diese Reduktion mit* $\to_{lz}$.
Eine Reduktion, die entweder eine Beta-Reduktions oder aus eine Case-Reduktions durchführt und sharing beachtet, sei als $\to_s$ *bezeichnet.*
Die Anwendungsordnung mit sharing bezeichnen wir als $\to_{aos}$.

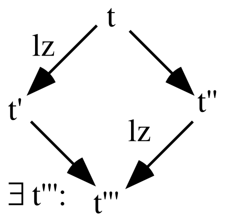
In der Implementierung werden dann als gleich markierte Terme auch gemeinsam benutzt, d.h. nur einmal gespeichert.

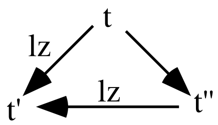
Lemma 13.3. *Gleich markierte Positionen liegen stets nebeneinander, d.h. an unabhängigen Positionen. Sie sind gleich, auch wenn man alle Markierungen mit beachtet.*

Beweis. Das folgt allein schon daraus, dass die als gleich markierten Positionen identische Ausdrücke sind, die nicht ineinander liegen können. Dass die Terme gleich sind, folgt daraus, dass sie stets gleich behandelt werden. □

In den folgenden Lemmas und Aussagen verwenden wir Terme mit sharing.

Lemma 13.4. *Wenn $t \rightarrow_{lz} t'$ und $t \rightarrow_s t''$, dann gilt entweder $t' = t''$, oder $t'' \rightarrow_{lz} t'$, oder es gibt ein t''', so daß $t'' \rightarrow_{lz} t'''$ und $t' \rightarrow_s t'''$.*



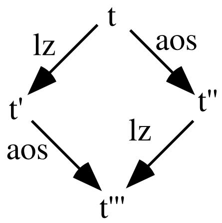


Beweis. Wir argumentieren in diesem Beweis mit gerichteten Graphen, simuliert mit Markierungen in Bäumen. Allerdings erfordert ein genauer Beweis eine besser handhabbare Formalisierung. Die hier gegebenen Begründungen sind eher eine Skizze.
i) Sei $t \equiv R[f\ t_1 \dots t_n]$ geschlossen und $f\ t_1 \dots t_n$ der Normalordnungs-Redex. Dann hat $(f\ t_1 \dots t_n)$ keine freien Variablen. Sei f definiert als $f\ x_1 \dots x_n = r$. Dann wird $f\ t_1 \dots t_n$ zu $r[t_1/x_1 \dots t_n/x_n]$ reduziert. Um alle Reduktionen anzudeuten, schreiben wir: $t \equiv C[f\ t_1 \dots t_n, \dots, f\ t_1 \dots t_n]$. Die lazy Reduktion reduziert alle gleichzeitig. D.h. $C[f\ t_1 \dots t_n, \dots, f\ t_1 \dots t_n] \xrightarrow{no} C[r[t_1/x_1 \dots t_n/x_n], \dots, r[t_1/x_1 \dots t_n/x_n]]$.
Wenn $t \rightarrow_s t''$ eine Reduktion ist, die nur von $f\ t_1 \dots t_n$ unabhängige Redexe reduziert, dann sind die Reduktionen vertauschbar.
Wenn die shared-Reduktion auch in einigen t_i gemacht wird, dann gilt: $t_i \rightarrow t'_i$, und $t''' = C'[r[t'_1/x_1 \dots \dots t'_n/x_n]$, wobei alle Reduktionen "parallel" in den als gleich markierten Termen gemacht werden, möglicherweise auch in C. Abhängig davon, ob x_i in r vorkommt oder nicht, gilt einer der beiden im Lemma genannten Fälle.
Wenn die shared-Reduktion oberhalb einer Variante des Ausdrucks $f\ t_1 \dots t_n$ gemacht wird, dann enthält t'' evtl. mehr Kopien dieses Terms, aber der Normalordnungsredex hat sich nicht geändert. Das Diagramm läßt sich schließen wie in den anderen Fällen.
ii) Wenn der NO-Redex ein `case`-Ausdruck ist, dann ist die Begründung analog.
□

Mit $\#LR(t)$ bezeichnen wir die Anzahl der Lazy-Reduktions-schritte des Terms t bis zum Erreichen einer WHNF. Mit $\#AO(t)$ die Anzahl der Reduktionsschritte in Anwendungsordnung (mit sharing). Wir schreiben $\#AO(t) = \infty$, wenn die Anwendungsordnung nicht terminiert.

Lemma 13.5. *Sei t ein Ausdruck. Wenn $t \to_{lz} t'$ und $\#AO(t) < \infty$, dann ist $\#AO(t') < \#AO(t)$.*

Beweis. Induktion nach $\#AO(t).Wenn\#$AO(t) $= 1$, dann gilt sowohl $t \to_{lz} t'$ als auch $t \to_{aos} t'$ und damit ist t' in WHNF.
Wenn $\#AO(t) = n > 1$, dann betrachte $t \to_{aos} t''$. Nach Lemma 13.4 gibt es drei Fälle:
Wenn $t'' = t'$, dann können wir Induktion verwenden.
Wenn $t'' \to_{lz} t'$, dann können wir Induktion verwenden und erhalten: $\#AO(t') < \#AO(t'') < \#AO(t)$.
Wenn $t'' \to_{lz} t'''$ und $t' \to_{aos} t'''$, dann gilt mit Induktion $\#AO(t''') < \#AO(t'')$, also auch $\#AO(t') = \#AO(t''') + 1 < \#AO(t'') + 1 = \#AO(t)$.



□

Aussage 13.6. $\#LR(t) \leq \#AO(t)$.

Beweis. Wir können annehmen, dass $\#AO(t) < \infty$. Mit Induktion nach der Anzahl der Schritte in AO zur WHNF. Wenn $\#AO(t) = 0$, dann ist t in WHNF, also ist auch $\#LR(t) = 0$. Sei $\#AO(t) = n > 0$. Dann betrachte $t \to_{aos} t''$. Sei $t \to_{lz} t'$. Es gilt $\#AO(t'') = n - 1$.
Wenn $t' = t''$, dann können wir Induktion verwenden.
Wenn $t'' \to_{lz} t'$, dann können wir ebenfalls Induktion verwenden und erhalten $\#LR(t'') \leq n - 1$, also $\#LR(t') \leq n - 2$, somit $\#LR(t) \leq n - 1 < \#AO(t) = n$.
Wenn $t'' \to_{lz} t'''$ und $t' \to_{aos} t'''$, dann gilt wegen Lemma 13.5, daß $\#AO(t''') < \#AO(t'') = n - 1$, somit $\#AO(t') \leq n - 1$, also mit Induktion $\#LR(t') \leq n - 1$.
Da $t \to_{no} t'$, erhalten wir $\#LR(t) \leq n$.

□

Theoretisch gilt: lazy evaluation (verzögerte Reduktion) ist noch nicht optimal in der Anzahl der Reduktionen. Der Grund liegt in der Ausführung der Ersetzung: Der Rumpf eines Superkombinators wird jedesmal kopiert, aber darin könnten sich auswertbare Ausdrücke befinden, die erst nach dem Kopieren des Rumpfs

ausgewertet werden. Analog werden beim Erzeugen von partiellen Anwendungen teilweise Reduktionen doppelt ausgeführt, obwohl dies nicht notwendig ist.
Es gibt Untersuchungen zu optimalen Reduktionsstrategien im Lambda-Kalkül. Diese Strategien können nur teilweise in die Implementierung von funktionalen Programmiersprachen Eingang finden.

Beispiel 13.7. `polynom x y = (x * x) + (y * y)`
Werte folgenden Ausdruck aus: `(\p .  (p 2) + (p 3))  (polynom 1)`

Man sieht, daß $(1 * 1)$ zweimal ausgewertet wird, obwohl dies eigentlich unnötig ist.

Die Standardkompilierung achtet nicht auf diese Optimierungsmöglichkeit. Den Optimierungsschritt, der zum Ziel hat, möglichst viele der reduzierbaren Ausdrücke bereits zur Kompilierzeit auszuwerten, nennt man "partielle Auswertung". Wie wir gesehen haben, ist die partielle Auswertung korrekt im Sinne der kontextuellen Äquivalenz.
Es gibt eine Optimierung, die auf ein besseres Kopieren des Funktionsrumpfes achtet und Ineffizienzen wie oben vermeidet ("fully lazy").
Beim Kompilieren hat die Anwendungsordnung den Vorteil, daß man zur Kompilierzeit für jede Funktion genau die Reihenfolge der Auswertungen vorhersagen kann und daß man keinen Test benötigt, ob Ausdrücke ausgewertet sind.

13.2 Sequentialität, Parallelität

Betrachtet man die Funktion $||$, das logische oder, dann beobachtet man, dass `True` $|| \perp$ zu `True` auswertet, während $\perp ||$ `True` nicht terminiert, obwohl man eigentlich sagen könnte, dass eigentlich nur `True` als Wert zurückkommen kann. Man kann auf die Idee kommen, dies zu verhindern, und zu versuchen, die scheinbar falsche Implementierung von $||$ zu verbessern. Allerdings werden wir zeigen, dass man keine Funktion definieren kann, die dieses Verhalten zeigt. Wenn man dieses Verhalten erzwingen will, dann muss man den ganzen Auswertungsprozess selbst in KFP (Haskell) implementieren und dann den linken und rechten Term parallel auswerten (d.h. irgendwie abwechselnd).

Aussage 13.8. *Man kann in KFP (Haskell, Gofer, Lambda-Kalkül,...) keine arithmetische Funktion $*$ definieren, die*

$$0 * \perp = \perp * 0 = 0$$

liefert, aber ansonsten das Produkt der Argumente als Resultat liefert.

Beweis. Es genügt dies für einen Datentyp mit Konstanten $0, 1$ zu zeigen und eine Multiplikationstabelle $0 * 1 = 1 * 0 = 0 * 0 = 0$, $1 * 1 = 1$. Zusätzlich soll $0 * \perp = \perp * 0 = 0$ aber auch $1 * \perp = \perp * 1 = \perp$ gelten.
Angenommen, man hat ein $f\ x\ y$ in KFP definiert, das dieses leistet. Dann betrachte die Auswertung $(f\ t_1\ t_2)$.

Die Normalordnung findet eine WHNF von $(f\ t_1\ t_2)$, falls eine existiert. Eine Eigenschaft der Normalordnung ist: wenn ein Unterausdruck reduziert wird, dann wird dieser mindestens solange reduziert, bis er selbst in WHNF ist. D.h., wenn man $f\ t_1\ t_2$ reduziert, dann wird irgendwann zum ersten mal entweder t_1 oder t_2 reduziert. Diese Entscheidung hängt aber nicht von der speziellen Form von t_1 oder t_2 ab. ObdA sei t_1 das erste Argument, das in Normalordnung reduziert wird. Dann ist $f \perp t_2 = \perp$, unabhängig vom Inhalt von t_2, denn wenn es einen anderen Wert hätte, würde die Normalordnung die Kopfnormalform nicht finden. Damit kann es eine solche Funktion nicht geben. □

Aussage 13.9. *Für jeden KFP+-Superkombinator f gilt:*
Entweder hat für jede Anzahl n und alle Argumente a_i der Term $f\ a_1 \dots a_n$ eine Abstraktion als WHNF (d.h. ist $\sim_c Y\ K$), oder es gibt eine Anzahl n von Argumenten, so dass eine der folgenden Aussagen gilt:

- $\forall a_1, \dots a_n : (f\ a_1 \dots\ a_n)\Uparrow$.
- *Es gibt ein Konstruktor c, so dass $\forall a_1, \dots a_n : f\ a_1 \dots\ a_n$ hat eine WHNF mit c als Top-Konstruktor.*
- *f ist strikt in mindestens einem Argument.*

Beweis. Wir nehmen an, dass $f \not\sim_c Y\ K$. Die Reduktion von $f\ x_1 \dots x_n$ kann folgendes ergeben:

- sie kann nicht terminieren oder einen Typfehler liefern. Dann gilt 13.9
- oder es ergibt sich ein Term der Form `(c t_1 ... t_m)`; dann gilt Fall 13.9.
- Sie kann einen Term der Form $\lambda y.t$ ergeben. Dann kann man noch ein Argument hinzufügen. Das Hinzufügen terminiert, denn $f \sim_c Y\ K$ war ausgeschlossen.
- Sie kann einen Term $R[x_i]$ ergeben, wobei R ein Reduktionskontext ist. Das ist Fall 13.9.

□

Beachte, dass der Fall $f \sim_c Y\ K$ bei polymorpher Typisierung nicht auftritt, da der Typ unendlich groß sein müßte. Die Anzahl n läßt sich bei polymorph getypten Superkombinatoren am Typ ablesen.

Literatur

[Bar84] H.P. Barendregt. *The Lambda Calculus. Its Syntax and Semantics.* North-Holland, Amsterdam, New York, 1984.