Der untypisierte Lambda-Kalkül

Der Lambda-Kalkül wurde zur Beschreibung des mathematischen Begriffs "Funktion" entwickelt.

Die Menge aller *Lambda-Ausdrücke* Exp ist folgendermaßen definiert:

- $x \in Exp$ wenn $x \in Id$; Id ist eine Bezeichner-Menge;
- $(e_1\, e_2) \in Exp$ wenn $e_1, e_2 \in Exp$ ("Funktionsaufruf");
- $(\lambda x.e) \in Exp$ wenn $x \in Id$ und $e \in Exp$ ("Funktionsabstraktion", x ist in e gebunden).

Jeder Ausdruck in Exp ist ein Programm mit wohldefinierter Semantik.

Die Regeln des Lambda-Kalküls

Zusatzdefinitionen (informal):

- $free(e)$ bezeichnet die Menge aller *freien Variablen* in e.
- $[e_1/x]e_2$ bezeichnet eine *Ersetzung* aller freien Vorkommen von $x \in Id$ in e_2 durch e_1.

Regeln legen die Semantik von Lambda-Ausdrücken fest:

- α-Konversion (Umbenennung):
  $\lambda x_1.e \Longleftrightarrow \lambda x_2.[x_2/x_1]e$, wobei $x_2 \in Id$ und $x_2 \notin free(e)$.
- β-Konversion (Anwendung): $(\lambda x.e_1)\, e_2 \Longleftrightarrow [e_2/x]e_1$.
- η-Konversion: $\lambda x.(e\, x) \Longleftrightarrow e$ wenn $x \in Id$ und $x \notin free(e)$.

Die gerichtete Variante der β- bzw. η-Konversion heisst β- bzw. η-Reduktion.

Normalform im Lambda-Kalkül

Ein Lambda-Ausdruck ist in *Normalform* wenn er mittels β- und η-Reduktionen nicht weiter reduziert werden kann.

Für einige Lambda-Ausdrücke, z.b. $(\lambda x.(x\,x))\,(\lambda x.(x\,x))$, gibt es keine Normalform.

Kein Lambda-Ausdruck kann in zwei verschiedene Normalformen konvertiert werden, wenn man Namensunterschiede aufgrund von α-Konversionen unberücksichtigt lässt.

Rekursion im Lambda-Kalkül

Jeder Lambda-Ausdruck e hat einen Fixpunkt e', so dass $(e\,e')$ zu e' konvertiert werden kann.

Beweis: Für e' nehmen wir $(Y\,e)$, wobei

$$Y \equiv \lambda f.((\lambda x.(f\,(x\,x)))\,(\lambda x.(f\,(x\,x)))).$$

Dann gilt:

$$\begin{aligned}(Y\,e) &= (\lambda x.(e\,(x\,x)))\,(\lambda x.(e\,(x\,x))) \\ &= e\,((\lambda x.(e\,(x\,x)))\,(\lambda x.(e\,(x\,x)))) \\ &= e\,(Y\,e)\end{aligned}$$

Jede rekursive Funktion kann leicht durch eine rekursionsfreie (und nicht-iterative) Funktion ersetzt werden:

- Ursprüngliche rekursive Funktion: $f \equiv \cdots f \cdots$
- Umgeschriebene rekursive Funktion: $f \equiv (\lambda f.\ \cdots f \cdots)\,f$
- Rekursionsfreie Definition: $f \equiv Y\,(\lambda f.\ \cdots f \cdots)$

Mächtigkeit des Lambda-Kalküls

Churchs These:

> *Genau jene Funktionen von den natürlichen Zahlen in die natürlichen Zahlen sind effektiv berechenbar, die im Lambda-Kalkül ausgedrückt werden können.*

“effektiven berechenbar” = Turing-vollständig

Alles Berechenbare ist auch im Lambda-Kalkül berechenbar

Typisierte Lambda-Ausdrücke

Eine Menge von Basistypen $BasTyp$ ist als gegeben angenommen. Die Menge aller Typen Typ ist induktiv definiert:

- $b \in Typ$ wenn $b \in BasTyp$;
- $t_1 \rightarrow t_2 \in Typ$ wenn $t_1, t_2 \in Typ$.

Im typisierten Lambda-Kalkül ist Id eine Menge typisierter Bezeichner der Form $x{:}t$ mit $t \in Typ$.

Die Menge der typisierten Lambda-Ausdrücke Exp ist induktiv definiert:

- $x{:}t \in Exp$ wenn $x{:}t \in Id$;
- $(e_1{:}t_2 \rightarrow t_1\ e_2{:}t_2){:}t_1 \in Exp$ wenn $e_1{:}t_2 \rightarrow t_1, e_2{:}t_2 \in Exp$;
- $(\lambda x{:}t_2.e{:}t_1){:}t_2 \rightarrow t_1 \in Exp$ wenn $x{:}t_2 \in Id$ und $e{:}t_1 \in Exp$.

Regeln des typisierten Lambda-Kalküls

α-Konversion:
$(\lambda x_1{:}t_2.e{:}t_1){:}t_2{\to}t_1 \Longleftrightarrow (\lambda x_2{:}t_2.[x_2/x_1]e{:}t_1){:}t_2{\to}t_1$,
wobei $x_2{:}t_2 \in Id$ und $x_2 \notin \mathit{free}(e)$.

β-Konversion:
$((\lambda x{:}t_2.e_1{:}t_1){:}t_2{\to}t_1\ e_2{:}t_2){:}t_1 \Longleftrightarrow [e_2/x]e_1{:}t_1$.

η-Konversion:
$(\lambda x{:}t_2.(e{:}t_2{\to}t_1\ x{:}t_2){:}t_1){:}t_2{\to}t_1 \Longleftrightarrow e{:}t_2{\to}t_1$
wenn $x{:}t_2 \in Id$ und $x \notin \mathit{free}(e)$.

Mächtigkeit des typisierten Kalküls

Für jeden Ausdruck im typisierten Lambda-Kalkül gibt es genau eine effektiv berechenbare Normalform. Folglich können (aufgrund der Unentscheidbarkeit des Halteproblems) nicht alle effektiv berechenbaren Funktionen dargestellt werden.

Im typisierten Lambda-Kalkül kann *kein* Fixpunkt-Operator definiert werden: In $(e\,e)$ müsste e sowohl einen Typ $t_2 \rightarrow t_1$ als auch t_2 haben.

Der Kalkül kann um δ-Regeln erweitert werden, die typisierte Versionen des Y-Kombinators implementieren:

$$(Y_t{:}(t{\rightarrow}t){\rightarrow}t\ e{:}t{\rightarrow}t){:}t \Longleftrightarrow (e{:}t{\rightarrow}t\ (Y_t{:}(t{\rightarrow}t){\rightarrow}t\ e{:}t{\rightarrow}t){:}t){:}t$$

Strukturierte Typen (1)

Im Prinzip können δ-Regeln die Eigenschaften des Lambda-Kalküls beliebig verändern.

δ-Regeln können auch verwendet werden, um weitere strukturierte Typen zum typisierten Lambda-Kalkül dazuzufügen.

Der Typ eines *Cartesischen Produkts* $(e_1{:}t_1, e_2{:}t_2)$ ist $t_1 \times t_2$. Folgende δ-Regeln definieren Cartesische Produkte:

$$
\begin{array}{rcl}
(\texttt{car}{:}(t_1 \times t_2) \rightarrow t_1\, (e_1, e_2){:}t_1 \times t_2){:}t_1 & \Rightarrow & e_1{:}t_1 \\
(\texttt{cdr}{:}(t_1 \times t_2) \rightarrow t_2\, (e_1, e_2){:}t_1 \times t_2){:}t_2 & \Rightarrow & e_2{:}t_2 \\
((\texttt{cons}{:}t_1 \rightarrow (t_2 \rightarrow (t_1 \times t_2))\, e_1{:}t_1){:}t_2 \rightarrow (t_1 \times t_2)\, e_2{:}t_2){:}t_1 \times t_2 & \Rightarrow & (e_1, e_2){:}t_1 \times t_2
\end{array}
$$

Strukturierte Typen (2)

Ein *Verbund* ("record") ist eine Menge indizierter Werte:

$$\{\texttt{a}{=}3, \texttt{b}{=}\texttt{true}, \texttt{c}{=}\texttt{"S"}\} : \{\texttt{a:Integer}, \texttt{b:Boolean}, \texttt{c:String}\}$$

Für die Auswahl eines Feldes gibt es eine δ-Regel:

$$\{l_1{=}e_1, \ldots, l_n{=}e_n\}{:}\{l_1{:}t_1, \ldots, l_n{:}t_n\}.l_i \Rightarrow e_i{:}t_i$$

Eine *Variante* ordnet genau einem Index einen Wert zu:

$$[\texttt{a}{=}3] : [\texttt{a:Integer}, \texttt{b:Boolean}, \texttt{c:String}]$$

Die einzige Operation ist die Mehrfachverzweigung:

$$\begin{array}{l}(\texttt{case}\ [l_i{=}e_i]{:}[l_1{:}t_1, \ldots, l_n{:}t_n]\ \texttt{of} \\ \quad l_1{\Rightarrow}f_1{:}t_1{\to}t, \ldots, l_n{\Rightarrow}f_n{:}t_n{\to}t){:}t \Rightarrow (f_i{:}t_i{\to}t\, e_i{:}t_i){:}t\end{array}$$

Logikprogrammierung

Definite Horn-Klauseln der Form

$$\forall x_1. \cdots \forall x_m. (A \vee \neg B_1 \vee \cdots \vee \neg B_n)$$

sind eine (mächtige) Untermenge der Prädikatenlogik erster Stufe. Die übliche Schreibweise ist

$$A \leftarrow B_1, \ldots, B_n.$$

Bedeutung: "Wenn es Belegungen für alle Variablen in den Atomformeln $A, B_1, \ldots, B_n$ gibt, so dass B_1 bis B_n alle zugleich wahr sind, dann ist auch A mit diesen Variablenbelegungen wahr."

Prolog

Ein definites Programm besteht aus einer Konjunktion von definiten Klauseln. Beispiel in Prolog:

```
append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

Ein Prologinterpreter versucht, ein Ziel (eine Anfrage) zu beweisen. Zum Beispiel liefert "`?- append([1,2,3], [4,5], Zs).`" die Antwort "`Zs=[1,2,3,4,5]`".

Mittels "Backtracking" können etwaige alternative Lösungen gefunden werden.

Typen und Logikprogramme

Typen in Prolog können auf dieselbe Weise wie im Lambda-Kalkül eingeführt werden: jeder Term bekommt einen Typ.

Wenn T_i die Menge der durch einen Typ t_i beschriebenen Terme bezeichnet, dann haben typisierte definite Hornklauseln die Form $\forall x_1 \in T_1. \cdots \forall x_m \in T_m.(A \vee \neg B_1 \vee \cdots \vee \neg B_n)$.

Jeden Typ t_i kann man auch als einstelliges Prädikat auffassen, das genau dann wahr ist, wenn sein Argument in T_i liegt. Damit kann man obige Klausel äquivalent auch in dieser Form schreiben: $A \leftarrow t_1(x_1), \ldots, t_m(x_m), B_1, \ldots, B_n$.

Ein Logikprogramm definiert ein Typsystem. Die Ausführung eines Programms entspricht einer Typüberprüfung.

Typisierte Logikprogramme

Die Sichtweise "Logikprogramm = Typsystem" ist problematisch, da nicht zwischen Typfehlern und Einschränkungen der Lösungsmenge unterschieden wird.

Mögliche Lösung: Es wird zwischen Typprädikaten und sonstigen Prädikaten unterschieden. Wenn ein sonstiges Prädikat nicht erfüllt ist, erfolgt Backtracking; wenn ein Typprädikat nicht erfüllt ist, wird eine Ausnahmebehandlung eingeleitet.

Durch syntaktische Einschränkungen sind auch stark typisierte Logikprogrammiersprachen möglich. Diese Typsysteme sind in der Regel polymorph.

Algebren in der Mathematik

Eine *universelle Algebra* ist ein Paar $\langle A, \Omega \rangle$, wobei A die Trägermenge und Ω ein System von Operationen auf A ist.

Der Typ einer Algebra beschreibt die Stelligkeiten und eine Menge von Gesetzen Δ das Verhalten der Operationen.

Beispiel: Der Ring über den ganzen Zahlen $\langle Z, \{+, \cdot, -, 0\} \rangle$ ist eine Algebra des Typs $\langle 2, 2, 1, 0 \rangle$ mit (u.a.) diesen Gesetzen:

$$(a + b) + c = a + (b + c) \qquad a + b = b + a$$
$$a + 0 = a \qquad a + (-a) = 0$$
$$(a \cdot b) \cdot c = a \cdot (b \cdot c)$$

Eine *Varietät* $V(\Omega, \Delta)$ ist eine Familie universeller Algebren mit gemeinsamem Ω und Δ. In *freien Algebren* in $V(\Omega, \Delta)$ gelten nur die aus Δ ableitbaren Gesetze.

Algebren zur ADT-Spezifikation

Die *Signatur* einer universellen Algebra besteht aus der Menge der Operationen und dem Typ der Algebra.

Eine Signatur entspricht der Schnittstelle eines ADT.

Spezifikation des ADT `WW` als Beispiel:

Operationen: $\Omega = \{\texttt{wahr}, \texttt{falsch}, \texttt{nicht}, \texttt{und}, \texttt{oder}\}$

Typ der Algebra: $\langle 0, 0, 1, 2, 2 \rangle$

Gleichungen:

$$\texttt{nicht(wahr)} = \texttt{falsch} \qquad \texttt{nicht(falsch)} = \texttt{wahr}$$
$$\texttt{und}(w,\texttt{wahr}) = w \qquad \texttt{oder}(w,\texttt{wahr}) = \texttt{wahr}$$
$$\texttt{und}(w,\texttt{falsch}) = \texttt{falsch} \qquad \texttt{oder}(w,\texttt{falsch}) = w$$
$$\texttt{und}(v,w) = \texttt{und}(w,v) \qquad \texttt{oder}(v,w) = \texttt{oder}(w,v)$$

Heterogene Algebren

Eine *heterogene Algebra* ist ein Tupel $\langle A_1, \ldots, A_n, \Omega \rangle$, wobei $A_1, \ldots, A_n$ Trägermengen und Ω Operationen sind.

Jeder Trägermenge A_i ist eine Sorte s_i zugeordnet.

Der Index jeder Operation beschreibt die Sorten der Operanden und des Ergebnisses.

Die *Signatur* besteht aus der Menge der Operationen sowie dem Typ und den Indizes aller Operationen.

Jede Sorte entspricht einem Typ in einer Programmiersprache.

Beispiel für heterogene Algebra (1)

Sorten: `WListe`, `WW`

Operationen:

$$
\begin{aligned}
\texttt{leerw} &: \texttt{WListe} \\
\texttt{cons} &: \texttt{WW} \times \texttt{WListe} \longrightarrow \texttt{WListe} \\
\texttt{car} &: \texttt{WListe} \longrightarrow \texttt{WW} \\
\texttt{cdr} &: \texttt{WListe} \longrightarrow \texttt{WListe} \\
\texttt{wahr} &: \texttt{WW} \\
\texttt{falsch} &: \texttt{WW} \\
\texttt{nicht} &: \texttt{WW} \longrightarrow \texttt{WW} \\
\texttt{und} &: \texttt{WW} \times \texttt{WW} \longrightarrow \texttt{WW} \\
\texttt{oder} &: \texttt{WW} \times \texttt{WW} \longrightarrow \texttt{WW}
\end{aligned}
$$

Beispiel für heterogene Algebra (2)

Gleichungen:

$$
\begin{aligned}
\texttt{car(cons(}w\texttt{, }l\texttt{))} &= w \\
\texttt{cdr(cons(}w\texttt{, }l\texttt{))} &= l \\
\texttt{cdr(leerw)} &= \texttt{leerw} \\
\texttt{nicht(wahr)} &= \texttt{falsch} \\
\texttt{nicht(falsch)} &= \texttt{wahr} \\
\texttt{und(}w\texttt{, wahr)} &= w \\
\texttt{und(}w\texttt{, falsch)} &= \texttt{falsch} \\
\texttt{und(}v\texttt{, }w\texttt{)} &= \texttt{und(}w\texttt{, }v\texttt{)} \\
\texttt{oder(}w\texttt{, wahr)} &= \texttt{wahr} \\
\texttt{oder(}w\texttt{, falsch)} &= w \\
\texttt{oder(}v\texttt{, }w\texttt{)} &= \texttt{oder(}w\texttt{, }v\texttt{)}
\end{aligned}
$$

Sorten als Parameter

`WListe` und `WW` können nur beschränkt als ADT bezeichnet werden, da diese Typen nur in einer gemeinsamen Datenkapsel spezifiziert werden können.

Im Endeffekt muss jedes ausreichend komplexe Programm in einer einzigen Datenkapsel dargestellt werden.

Als Lösung bieten sich parametrisch polymorphe Typen an: Sorten werden als Parameter an andere Algebren übergeben.

Diese Lösung funktioniert immer, wenn keine zyklischen Typabhängigkeiten bestehen.

Beispiel für Parameter-Sorten

Sorten: `Liste`

Parameter-Sorten: `Element`

Operationen:

`leer : Liste`

`cons : Element` $\times$ `Liste` $\rightarrow$ `Liste`

`car : Liste` $\rightarrow$ `Element`

`cdr : Liste` $\rightarrow$ `Liste`

Gleichungen:

`car(cons(`e`, `l`)) = `e

`cdr(cons(`e`, `l`)) = `l

`cdr(leer) = leer`

Prozessalgebra

Sie dient zur Spezifikation nebenläufiger Prozesse.

Atomare Aktionen (Befehle einer virtuellen Maschine) werden mittels $*$ (Hintereinanderausführung), $||$ (Parallelausführung) und $+$ (alternative Ausführung) verknüpft.

Äquivalenz wird über algebraische Gesetze ausgedrückt:

$$x + y = y + x \qquad (x + y) + z = x + (y + z)$$
$$x + x = x \qquad (x + y) * z = x * z + y * z$$
$$(x * y) * z = x * (y * z) \qquad x * \varepsilon = x$$
$$\varepsilon * x = x \qquad x \,||\, y = y \,||\, x$$
$$(x + y) \,||\, z = x \,||\, z + y \,||\, z \qquad (x \,||\, y) \,||\, z = x \,||\, (y \,||\, z)$$
$$x \,||\, \varepsilon = x$$