
Konzepte höherer Programmiersprachen

Kurt Sieber

Fachbereich Mathematik/Theoretische Informatik
Universität Siegen

Vorlesung vom 04.05.2005

Vom λ -Kalkül zur Programmiersprache

- Im reinen ungetypten λ -Kalkül gibt es *nur* Funktionen
- Einzige Rechenregel ist die β -Reduktion
- Insbesondere gibt es keine Typfehler (auch nicht zur Laufzeit)

Aus theoretischer Sicht reicht das, um alle berechenbaren Funktionen (auf Church numerals) zu implementieren.

Aber für eine echte Programmiersprache (auch für eine rein funktionale) benötigt man zusätzliche Sprachkonstrukte.

Insbesondere ist das Fehlen jeglicher Typfehler im ungetypten λ -Kalkül eher von Nachteil, weil dann auch jegliche Sicherheit beim Programmieren fehlt.

Vom λ -Kalkül zur Programmiersprache

Um den ungetypten λ -Kalkül zu einer Programmiersprache auszubauen, muss man zumindest folgende Sprachkonzepte aufnehmen:

1. *Basisdaten* wie Zahlen, boolesche Werte, strings, ...
2. *Kontrollstrukturen* wie Verzweigung und Rekursion
3. *Datenstrukturen* wie Tupel, Listen, Bäume, ...
4. eine *Auswertungsstrategie*, z.B. call-by-value
5. *Typüberprüfung* entweder zur Laufzeit (Lisp/Scheme) oder zur Compilezeit (SML/O'CamL)

Vom λ -Kalkül zur Programmiersprache

Basisdaten, Kontrollstrukturen und Datenstrukturen kann man zwar (ähnlich wie die Church numerals) im ungetypten λ -Kalkül simulieren, aber:

- Eine solche Simulation ist ineffizient.
- Sie ist nur eine theoretische Spielerei, die nichts zum Verständnis der neuen Sprachkonzepte beiträgt.
- Es fehlt nach wie vor jegliche Typüberprüfung (weil dann auch boolesche Werte, Paare und Listen nichts anderes als Funktionen sind).

Deshalb nehmen wir solche Sprachkonzepte als *explizite Erweiterungen* (und nicht als syntaktischen Zucker) zum λ -Kalkül hinzu.

Vom λ -Kalkül zur Programmiersprache

Zunächst: Dynamisch getypte Sprachen (Lisp/Scheme)

- Vorteile:

Sehr einfache und einheitliche Syntax (und Semantik)

Die Kernsprache (engl.: core language) ist aus wenigen syntaktischen Konstrukten aufgebaut. Alles weitere kann man als syntaktischen Zucker einführen.

Man hat große Freiheit (Flexibilität) beim Programmieren, weil ein statisches Typsystem fehlt.

- Nachteile:

Zu einheitliche Syntax, mitunter schlecht lesbar.

Zu viel Freiheit beim Programmieren: Viele Fehler fallen erst zur Laufzeit auf, weil ein statisches Typsystem fehlt. Dann ist es meist schwieriger, den Fehler zu lokalisieren.

Vom λ -Kalkül zur Programmiersprache

Die einzigen Ausdrücke (auch *S-expressions* genannt) in Lisp/Scheme sind

- *Atome* (Konstanten und symbolische Atome)
- *Paare* (auch *dotted pairs* oder *cons-Zellen* genannt)

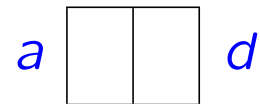
Zu den Konstanten zählen Zahlen, boolesche Werte, strings und die leere Liste.

Symbolische Atome dienen entweder als Namen für Werte (die man wie im λ -Kalkül als *Variablen* bezeichnet), oder tatsächlich als *Symbole* (vergleichbar mit den Konstruktoren in ML).

Vom λ -Kalkül zur Programmiersprache

Paare haben die Form $(e_1 . e_2)$, d.h. die Komponenten eines Paares werden durch einen Punkt getrennt ('dotted pair').

Graphisch werden Paare oft als 'cons-Zellen' dargestellt:



Den linken und rechten Teil eines Paares nennt man (aus historischen Gründen) *address* part und *decrement* part.

Ein Paar kann man mit der Funktion *cons* (= 'construct') konstruieren, z.B. liefert $(cons\ 1\ 2)$ das Paar $(1 . 2)$.

Auf die Komponenten eines Paares kann man mit den Funktionen *car* ('contents of address register') und *cdr* ('contents of decrement register') zugreifen.

Vom λ -Kalkül zur Programmiersprache

Listen erhält man durch Schachtelung von Paaren.

Eine Liste hat die Form $(e_1 \dots e_n)$ mit $n \geq 0$.

Diese Schreibweise ist syntaktischer Zucker für

$$(cons\ e_1\ \dots\ (cons\ e_n\ ()))\ \dots)$$

Auf die Komponenten einer Liste kann man also durch Schachtelung von *cars* und *cdrs* zugreifen.

Da man solche Zugriffe häufig benötigt, gibt es dafür syntaktischen Zucker, z.B.:

$$(caddr\ e)\ \text{für}\ (cdr\ (cdr\ (cdr\ e)))$$
$$(caddr\ e)\ \text{für}\ (car\ (cdr\ (cdr\ e)))$$

Listen sind die wichtigste Datenstruktur in Lisp/Scheme:

Lisp bedeutet 'List Processor'.

Vom λ -Kalkül zur Programmiersprache

Wie schreibt man Programme, wenn man nur Atome und Paare (und Listen) hat? Wo ist die Verbindung zum λ -Kalkül?

Die Antwort liegt in der Arbeitsweise des Interpreters (= *read-eval-print-loop* oder kurz *repl*). Der Interpreter fasst jede Liste

$$(e_1 \dots e_n) \quad (n \geq 1)$$

als Applikation von e_1 (dem *car* der Liste) auf die Restliste $(e_2 \dots e_n)$ (dem *cdr* der Liste) auf, z.B.:

- (+ 2 3) Applikation von + auf die Liste (2 3), liefert 5
- (2 3) Applikation von 2 auf die Liste (3), Laufzeittypfehler
- (2) Applikation von 2 auf die Liste (), Laufzeittypfehler

Als Auswertungsstrategie wird call-by-value benutzt, d.h. sowohl der Funktionsteil e_1 als auch die Argumente e_2, \dots, e_n werden *vor* der eigentlichen Funktionsanwendung ausgewertet.
