
Theorie der Programmierung I

Kurt Sieber

Fachbereich Mathematik/Theoretische Informatik
Universität Siegen

Vorlesung vom 27.10.2004 (Stand: 28.10.2004)

Typsystem

Statt $[] \triangleright e :: \tau$ schreiben wir kürzer $e :: \tau$ (lies: “ e hat Typ τ ”).

Definition 1.2 Die *Ordnung* $ord(\tau)$ eines Typs τ ist induktiv definiert durch

$$\begin{aligned} ord(\beta) &= 0 \\ ord(\tau_1 * \dots * \tau_n) &= \max\{ord(\tau_1), \dots, ord(\tau_n)\} \\ ord(\tau_1 \rightarrow \tau_2) &= \max\{ord(\tau_1) + 1, ord(\tau_2)\} \end{aligned}$$

Ein Typ *höherer Ordnung* oder *höherer Stufe* ist ein Typ der Ordnung ≥ 2 . Diese Begriffe überträgt man von Typen auf Ausdrücke bzw. Funktionen, d.h. man spricht von Funktionen erster, zweiter, dritter oder höherer Stufe, wenn sie einen entsprechenden Typ haben.

Typsystem

Beispiele:

- $ord(\mathbf{int} \rightarrow \mathbf{int}) = \max\{ord(\mathbf{int}) + 1, ord(\mathbf{int})\} = \max\{1, 0\} = 1$, also sind \sim – und *square* Funktionen erster Stufe.
- $ord(\mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{int}) = \max\{ord(\mathbf{int}) + 1, ord(\mathbf{int} \rightarrow \mathbf{int})\} = 1$, also sind $+$, $-$, ... Funktionen erster Stufe.
- $ord(\mathbf{int} * \mathbf{int} \rightarrow \mathbf{int}) = \max\{ord(\mathbf{int} * \mathbf{int}) + 1, ord(\mathbf{int})\} = 1$, also ist z.B. $\lambda z : \mathbf{int} * \mathbf{int}. \#1 z$ eine Funktion erster Stufe.
- $ord((\mathbf{int} \rightarrow \mathbf{int}) \rightarrow \mathbf{int}) = \max\{ord(\mathbf{int} \rightarrow \mathbf{int}) + 1, ord(\mathbf{int})\} = \max\{2, 0\} = 2$, also ist z.B. $\lambda f : \mathbf{int} \rightarrow \mathbf{int}. f 0$ eine Funktion zweiter und damit höherer Stufe.

Syntaktischer Zucker

Bisher: KFG und Typsystem für den **Kern** der Programmiersprache (engl. **core language**)

Jetzt: Einführung weiterer syntaktischer Konstrukte als abkürzende Schreibweisen.

Diese werden als **syntaktischer Zucker** oder **abgeleitete Formen** (engl. **derived forms**) bezeichnet.

Vorteil: Die Kernsyntax bleibt klein, Beweise für die Kernsyntax (oft durch Fallunterscheidung über alle Produktionen der KFG) werden nicht zu lang. Die bewiesenen Sätze übertragen sich meist 'automatisch' auf den syntaktischen Zucker.

Dies lässt sich auch bei der Implementierung einer Programmiersprache ausnutzen: Zuerst Übersetzung des syntaktischen Zuckers in die Kernsyntax, dann Typüberprüfung, dann Codegenerierung.

Kann aber zu unverständlichen (Typ-)Fehlermeldungen führen!

Syntaktischer Zucker

Mehrstellige λ -Abstraktion:

$\lambda(id_1, \dots, id_n) : \tau_1 * \dots * \tau_n. e$

steht für

$\lambda id : \tau_1 * \dots * \tau_n. \mathbf{let} \ id_1 = \#1 \ id \ \mathbf{in} \ \dots \ \mathbf{let} \ id_n = \#n \ id \ \mathbf{in} \ e$

Dabei muss gelten: id_1, \dots, id_n verschieden und id **neuer** Name (der nicht in e und unter id_1, \dots, id_n) vorkommt.

Beispiel:

$\lambda(x, y) : \mathbf{int} * \mathbf{int}. * \ x \ y$

steht für

$\lambda z : \mathbf{int} * \mathbf{int}. \mathbf{let} \ x = \#1 \ z \ \mathbf{in} \ \mathbf{let} \ y = \#2 \ z \ \mathbf{in} \ * \ x \ y$

Syntaktischer Zucker

Mehrstelliges let:

let $(id_1, \dots, id_n) = e_1$ **in** e_2

steht für

let $id = e_1$ **in let** $id_1 = \#1 id$ **in ... let** $id_n = \#n id$ **in** e_2

mit den gleichen Bedingungen wie oben

Beispiel:

let $(x, y) = (y, x)$ **in** ...

steht für

let $z = (y, x)$ **in let** $x = \#1 z$ **in let** $y = \#2 z$ **in** ...

(bewirkt eine 'Vertauschung' von x und y)

Konjunktion und Disjunktion

$e_1 \ \&\& \ e_2$

steht für

if e_1 **then** e_2 **else** *false*

und

$e_1 \ || \ e_2$

steht für

if e_1 **then** *true* **else** e_2

Syntaktischer Zucker

Infixschreibweise für binäre Operatoren

$e_1 \text{ op } e_2$

steht für

$\text{op } e_1 \ e_2$

falls $\text{op} \in \{=, <, >, \leq, \geq, +, -, *, /, \mathbf{mod}\}$

z.B. $x + 2$ für $+ x 2$

Syntaktischer Zucker

Konkrete Syntax

- Auch der syntaktische Zucker ist zunächst nur abstrakte Syntax.
- Für die konkrete Syntax vereinbaren wir:
 - Die Applikation bindet am stärksten
 - dann folgen $*$, $/$ und **mod**
 - dann $+$ und $-$
 - dann die Vergleichsoperatoren $=$, $<$, $>$, \leq und \geq
 - dann die neuen Schreibweisen $\&\&$ und $\|\|$
 - dann **if** _ **then** _ **else** _ und **let** _ **in** _
 - und am schwächsten bindet die λ -Abstraktion.

Außerdem müssen jetzt binäre Operatoren in Klammern stehen, wenn sie **nicht** in der Infixschreibweise verwendet werden, z.B. $(+)$ x 1 oder $(+)$ 1 .

- In Zweifelsfällen setzen wir sowieso Klammern!

Syntaktischer Zucker

Abgeleitete Typregeln

- Syntaktischer Zucker kann (jedes Mal) in die Kernsyntax übersetzt und auf Wohlgetyptheit überprüft werden.
- Alternative: Durch eine **einmalige** Übersetzung leitet man sich eigenständige Typregeln für den syntaktischen Zucker her.
- Diese nennt man **abgeleitete Regeln** (engl. **derived rules**).

Beispiel: Typregel für die mehrstellige λ -Abstraktion

$$\text{(ABSTR-n)} \quad \frac{\Gamma[\tau_1/id_1] \dots [\tau_n/id_n] \triangleright e :: \tau'}{\Gamma \triangleright \lambda(id_1, \dots, id_n) : \tau_1 * \dots * \tau_n. e :: \tau_1 * \dots * \tau_n \rightarrow \tau'}$$