

1 Primitiv rekursive Funktionen

Die primitiv rekursiven Funktionen sind definiert, als die kleinste Menge von Funktionen die für alle $i, n \in \mathbb{N}$ die Grundfunktionen *konstante Nullfunktion*, *Projektion*, *Nachfolgerfunktion* enthält und unter den Operationen *Komposition* und *primitive Rekursion* abgeschlossen ist. Diese sind für beliebige $i, n \in \mathbb{N}$ folgendermaßen definiert:

$$\begin{aligned} \mathbf{Z\ n}: \mathbb{N}^n &\rightarrow \mathbb{N} \\ \mathbf{Z\ n}(x_1, \dots, x_n) &= 0 \end{aligned}$$

$$\begin{aligned} \mathbf{S}: \mathbb{N} &\rightarrow \mathbb{N} \\ \mathbf{S}(x) &= x + 1 \end{aligned}$$

$$\begin{aligned} \mathbf{Pi\ n\ i}: \mathbb{N}^n &\rightarrow \mathbb{N} \\ \mathbf{Pi\ n\ i}(x_1, \dots, x_i, \dots, x_n) &= x_i \end{aligned}$$

Sei h eine primitiv rekursive Funktion mit Stelligkeit m und seien $g_1 \dots g_m$ primitiv rekursive Funktionen mit Stelligkeit n . Dann ist Komposition $f := \mathbf{C}(\mathbf{h}, (\mathbf{g}_1, \mathbf{g}_2, \dots, \mathbf{g}_m))$

$$\begin{aligned} f: \mathbb{N}^n &\rightarrow \mathbb{N} \\ f(x_1, \dots, x_n) &= h(g_1(x_1, \dots, x_n), g_2(x_1, \dots, x_n), \dots, g_m(x_1, \dots, x_n)) \end{aligned}$$

eine primitiv-rekursive Funktion.

Sei g ein n -stellige und h eine $n + 2$ -stellige primitiv rekursive Funktion. Dann ist die primitive Rekursion $f := P(g, h)$

$$\begin{aligned} f: \mathbb{N}^{n+1} &\rightarrow \mathbb{N} \\ f(0, x_1, \dots, x_n) &= g(x_1, \dots, x_n) \\ f(y + 1, x_1, \dots, x_n) &= h(f(y, x_1, \dots, x_n), y, x_1, \dots, x_n) \end{aligned}$$

eine primitiv rekursive Funktion.

1.1 Beispiel

$$\begin{aligned} plus: \mathbb{N}^2 &\rightarrow \mathbb{N} \\ plus(0, x_1) &= x_1 = \mathbf{Pi\ 1\ 1}(x_1) \\ plus(y + 1, x_1) &= plus(y, x_1) + 1 = \mathbf{C}(\mathbf{S}, (\mathbf{Pi\ 3\ 1}))(plus(y, x_1), y, x_1) \end{aligned}$$

Es gilt also: $plus = P (Pi\ 1\ 1, C (S, (Pi\ 3\ 1)))$.

Ein entsprechender Funktionsaufruf dieser Funktion sieht im Interpreter folgendermaßen aus:

```
P (Pi 1 1, C (S, (Pi 3 1))) (3,5)
```

Da zusätzlich *let-bindings* erlaubt sind, lässt sich der Code auch übersichtlicher schreiben:

```
let idOnN = Pi 1 1
let sucOffFirst = C (S, (Pi 3 1))
let plus = P (idOnN, sucOffFirst)
plus (3,5)
```

2 Definition der Sprache

nat	::=	[0-9]+	<i>Natural Numbers</i>
id	::=	[a-zA-Z][a-zA-Z0-9]*	<i>Identifier</i>
prek	::=	'Z' num	<i>Arity</i>
		'Pi' num num	<i>Arity and Index</i>
		'Suc'	<i>Suc $x = x + 1$</i>
		'P' '(' prek ' ',' prek' ')'	<i>Base Case and Inductive Case</i>
		'C' prek prektuple	
prek'	::=	prek	
		id	
prekTuple	::=	()	
		'(' prek '(' ',' prek ')' * ')'	
natTuple	::=	()	
		'(' nat '(' ',' nat ')' * ')'	
Assignment	::=	'let' id '=' prek	
FunCall	::=	id natTuple	
		prek natTuple	
Programm	::=	Assignment* FunCall*	

Literatur

- [1] Benjamin Hodgson. *Is it possible to "bake dimension into a type" in haskell?* <https://softwareengineering.stackexchange.com/questions/276867/is-it-possible-to-bake-dimension-into-a-type-in-haskell>. 2015.
- [2] A Martin-Pizarro. *Logik für Studierende der Informatik Kurzschrift*. 2019.
- [3] tommyengstrom. *Codemirror-Elm*. <https://github.com/tommyengstrom/codemirror-elm>. 2013.

- [4] Wikipedia. *Primitive recursive function* — *Wikipedia, The Free Encyclopedia*. <http://en.wikipedia.org/w/index.php?title=Primitive%20recursive%20function&oldid=1042441754>. [Online; accessed 27-October-2021]. 2021.