Lambda-Kalkül

Dr. Eva Richter

27.April 2012

Historisches

- um 1200 Einführung des Zahlensystems in Europa u.a. durch Fibonacci; ist charakterisiert durch ein kleine Menge von Ziffern, deren Wert von der Position in der Zahl abhängt
- Zahlsystem kam über Arabien aus Indien
- ca. 1580 wurden von Francois Viète erstmals Variablen als Platzhalter und Zeichen für Operationen zur Notation von Ausdrücken und Gleichungen verwendet
- ca. 1930 entwickelte Alonzo Church eine Schreibweise für beliebige Funktionen
- führte den Formalismus als funktionale Basis der Mathematik ein
- in den 1960-er Jahren wurde der λ-Kalkül als vielseitiges Werkzeug der Informatik von Leuten wie McCarthy, Strachey, Landin und Scott „wiederentdeckt“

Notationen in Programmiersprachen

- 1936-1950 wurden verschiedene Arten der Zahldarstellung ausprobiert
- Vietas Schreibweise für Ausdrücke war die große Innovation in FORTRAN (Backus 1953), die Assembler ablöste
- 1960 veröffentlichte McCarthy die Listenverarbeitungssprache Lisp, die an den λ-Kalkül erinnert
- heutige Programmiersprachen (z.B. Java, C++) trennen üblicherweise primitive Datentypen und Funktionen (Methoden)
- Linie von Lisp führte zu ML und Haskell, die keine objektorientierten Aspekte haben
- OCaml als ML-Dialekt ist eine der wenigen Sprachen, die beide Ansätze kombiniert

Ausdrücke im λ-Kalkül

- λ-Kalkül ist effiziente Schreibweise für Funktionen
- Ausdrücke werden in strikter Präfix-Form geschrieben, d.h es gibt weder Infix- noch Postfixoperatoren wie $+$ oder ^{2}
- Funktionen und Argumente werden nebeneinander geschrieben, ohne Klammern
- bei mehr als einem Argument, wird alles aneinandergereiht, z.B. $+\,3\,x$ statt $x+3$, $*x\,x$ statt x^2 und $+\,(sin\,x)\,4$

Ausdrücke im λ-Kalkül

- bei Ausdrücken, die eine Variable x enthalten, ist die Beziehung zwischen konkreten Werten und Wert des Ausdrucks eine Funktion; mathematisch $f(x) = 3x$ oder $x \mapsto 3x$
- λ-Ausdrücke brauchen keine Namen für Funktionen aus $f(x) = 3x$ wird $\lambda x. * 3\,x$
- λ macht klar, dass die folgende Variable nicht Teil eines Ausdrucks, sondern formaler Parameter einer Funktionsdeklaration ist, Punkt nach dem Parameter ist der Beginn der Funktionsbeschreibung

PASCAL	`function` f	$(x : int) : int$	`begin`	$f = 3 * x$ `end;`
	λ	x	.	$*3x$
Lisp	*lambda*	(x)		$(*3x)$

Iteration von Funktionen

- jede Funktion in λ-Schreibweise kann als Ausdruck verwendet werden,
- $(\lambda x. * 3\,x)\,4$ ist Anwendung der Funktion auf $x = 4$
- Klammern begrenzen die Definition: $\lambda x. * 3\,x\,4$ entspricht $3 * x * 4$ (falls $*$ dreistellig wäre, sonst sinnlos)
- Abkürzungen für Bequemlichkeit: Ist $F := \lambda x. * 3\,x$, kann man $F\,4$ schreiben anstelle von $(\lambda x. * 3\,x)\,4$
- Funktionskörper enthält eine Funktion
  Beispiel: $N := \lambda y.(\lambda x. * y\,x)$, dann ist $N\,3$ wieder $\lambda x. * 3\,x$, d.h. $N\,3$ verhält sich wie F
- um zu betonen, dass 3 zuerst verwendet wird, schreibt man $(N\,3)\,4$ für gleichzeitige Auswertung $N\,3\,4$

Offizielle Definition

Definition

Ein λ-Term wird konstruiert durch die folgende Grammatik:

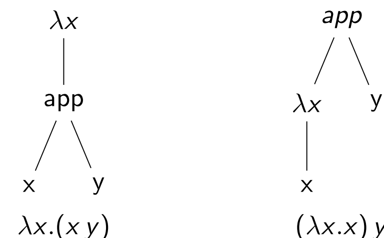
$$M ::= c|x|M\,M|\lambda x.M$$

wobei c Konstanten sind wie z.B. $1, 2, \ldots +, *$ und x eine (von unendlich vielen) Variablen.
Ein Ausdruck ohne Konstanten heißt reiner λ-Term.

Grammatik ist nicht eindeutig

$$M ::= c|x|M\,M|\lambda x.M$$

Der Term $\lambda x.x\,y$ kann auf zwei Arten gelesen werden:



Applikation ist linksassoziativ, d.h.

$E_1 E_2 E_3 \ldots E_n$ wird ausgewertet als $(\ldots(E_1 E_2) E_3) \ldots E_n)$

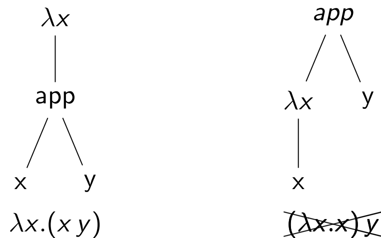
Abstraktion ist rechtsassoziativ, d.h.

$\lambda x.\lambda y.\lambda z.\,x\;y\;z$ wird ausgewertet als $\lambda x.(\lambda y.(\lambda z.\,x\;y\;z))$

Grammatik ist nicht eindeutig

$$M ::= c|x|M\,M|\lambda x.M$$

Der Term $\lambda x.x\,y$ kann auf zwei Arten gelesen werden:



Applikation ist linksassoziativ, d.h.

$E_1 E_2 E_3 \ldots E_n$ wird ausgewertet als $(\ldots(E_1 E_2)E_3)\ldots E_n)$

Abstraktion ist rechtsassoziativ, d.h.

$\lambda x.\lambda y.\lambda z.\,x\ y\ z$ wird ausgewertet als $\lambda x.(\lambda y.(\lambda z.\,x\ y\ z))$

Reduktion

- einzige Rechenregel ist Reduktion(β-Reduktion)
- beschreibt, wie formale Parameter durch tatsächliche ersetzt werden
- wird nur gebraucht, wenn ein Term auf einen anderen angewendet wird

$$(\lambda x. * 3\,x)\,4 \to_\beta *3\,4$$
$$(\lambda y.y\,5)(\lambda x. * 3\,x) \to_\beta (\lambda x. * 3\,x)5 \to_\beta *3\,5$$

- mit manchen Ausdrücken kann man das unendlich oft machen:

$$\Omega = (\lambda x.x\,x)(\lambda x.x\,x)$$

- ein Term ist in Normalform, falls keine weiteren Reduktionen möglich sind
- nicht jeder Term hat eine Normalform(z.B. Ω)

Konfluenz

- in manchen Fällen gibt es mehrere Möglichkeiten für eine Reduktion
- damit der Kalkül sinnvoll ist, sollte die Reihenfolge der Reduktionen keine Rolle spielen

Satz (Church-Rosser)

Falls ein Term M in einer endlichen Anzahl von Schritten zu einem Term N oder zu einem Term P reduziert werden kann, dann existiert ein Term Q zu dem sowohl N als auch P reduziert werden können.

Warnung: Nicht jede Auswertungsstrategie führt zur Normalform! Siehe Aufgabe 3.1.1.

Äquivalenz von λ-Termen

Folgerung

Jeder Term hat höchstens eine Normalform.

Beweis: Seien N und P zwei Normalformen zum Term M. Nach Church-Rosser existiert ein Q mit $P \to_\beta^* Q$ und $N \to_\beta^* Q$. Da P und N in NF folgt $P \equiv N \equiv Q$. □

- Terme s und t, die sich nur durch Namen der Variablen unterscheiden werden als äquivalent betrachtet: $s \equiv t$
- Terme s und t, die durch Umbenennung(α-Konversion) der Variablen und β-Reduktion auseinander hervorgehen, heißen λ-gleich: $s \approx t$.

Beispiele

$I = \lambda x.x$		Identität
$K = \lambda x.\lambda y.x$	$Kc = \lambda y.c$	Konstante
$B = \lambda x.\lambda y.\lambda z.x(y(z))$	$B\,f\,g = \lambda z.f(g(z))$	Komposition
$S = \lambda x.\lambda y.\lambda z.(xz)(yz)$		Substitution

$$
\begin{aligned}
SKK =& [\lambda x.\lambda y.\lambda z.(xz)(yz)]KK \\
=& [\lambda y.\lambda z.(Kz)(yz)]K \\
=& \lambda z.[(Kz)(Kz)] \quad K = \lambda x.\lambda y.x, Kz = \lambda y.z \\
=& \lambda z.[(\lambda y.z)(\lambda y.z)] \\
=& \lambda z.z
\end{aligned}
$$

Freie Variablen

- alle Namen sind lokale Definitionen
- Variable x in $\lambda x.x$ heißt gebunden, da sie im Körper der Funktionsdefinition, die mit λx beginnt, steht
- eine Variable, der kein zugehöriges λ vorangeht, heißt frei, z.B. y in $\lambda x.xy$
- in $(\lambda x.x)(\lambda y.yx)$ ist das x im linken Ausdruck gebunden, im zweiten Teil ist y gebunden und x ist frei, es ist vollkommen unabhängig von dem x im linken Teil

Definition

Eine Variable x ist frei in folgenden Fällen:

1. *x ist frei im Ausdruck x*
2. *x ist frei in $\lambda y.E$ falls x frei im Ausdruck E vorkommt und $y \neq x$*
3. *x ist frei in E_1E_2, falls x frei in E_1 oder E_2*

Gebundene Variablen und Substitution

Definition

Eine Variable x ist in folgenden Fällen gebunden

1. *x ist gebunden in $\lambda x.E$ und x ist frei in E*
2. *x ist gebunden in E_1E_2, falls x gebunden in E_1 oder gebunden in E_2.*

Eine Variable kann also in einem Ausdruck sowohl frei als auch gebunden sein!

1. Für Identitätsfunktion I ergibt $II \equiv (\lambda x.x)(\lambda x.x)$. Man kann auch $II \equiv (\lambda x.x)(\lambda z.z)$ schreiben und reduziert durch $[\lambda z.z/x]x$ zu $\lambda z.z \equiv I$
2. Vorsicht beim Substituieren, freie und gebundene Vorkommen dürfen nicht verwechselt werden

Beispiel Substitution in $(\lambda x.(\lambda y.xy))y$

das linke y ist gebunden, das rechte ist frei

falsch: ~~$\lambda y.yy$~~

richtig benenne gebundenes y in t um:

$$\lambda x.(\lambda t.tx)y \rightarrow_\beta \lambda t.yt$$

- wird $\lambda x.E_1$ auf E_2 angewendet, werden alle freien Vorkommen von x in E_1 durch E_2 ersetzt
- käme dabei eine freie Variable aus E_2 in einen Ausdruck, wo diese Variable gebunden ist, wird die gebundene Variable vorher umbenannt
- in
$$(\lambda x.(\lambda y.(x(\lambda x.xy))))y$$
ersetzt man y im Innern durch t zu
$$(\lambda x.(\lambda t.(x(\lambda x.xt))))y \rightarrow_\beta (\lambda t.(y(\lambda x.xt)))$$

Natürliche Zahlen–Church Numerals

- können durch 0 und Nachfolgerfunktion dargestellt werden: $zero, suc(zero), suc(suc(zero))\dots$
- Null wird definiert als $\overline{\mathbf{0}} \equiv \lambda s.(\lambda z.z)$
- weitere Zahlen: $\overline{\mathbf{1}} \equiv \lambda s.\lambda z.s(z), \overline{\mathbf{2}} \equiv \lambda s.\lambda z.s(s(z)), \overline{\mathbf{3}} \equiv \lambda s.\lambda z.s(s(s(z)))$ usw.
- Nachfolgerfunktion $S \equiv \lambda w.\lambda y.\lambda x.y(wyx)$ angewendet auf $zero$:
  $\lambda w\ \lambda y.\lambda x.(y(\ w\ yx))\ (\lambda s.(\lambda z.z)) \rightarrow$
  $\lambda y.\lambda x.(y(\ \lambda s\ .(\lambda z.z))\ y\ x) \rightarrow$
  $\lambda y.\lambda x.(y(\ \lambda z.z\ )\ x\ ) \rightarrow \lambda y.\lambda x.(y(x)) \equiv \overline{\mathbf{1}}$
- $(\lambda w.\lambda y.\lambda x.y(wyx))(\lambda s.\lambda z.s(z)) \rightarrow$
  $(\lambda y.\lambda x.y((\lambda s.\lambda z.s(z))yx)) \rightarrow$
  $(\lambda y.\lambda x.y(\lambda z.y(z))x) \rightarrow (\lambda y.\lambda x.y(y(x)))$

Addition

- grüner Teil in $\overline{\mathbf{1}} \equiv \lambda s.\lambda z.\ s(z)$ ist Anwendung von s auf z
- $2+3$ bedeutet 2-malige Anwendung der Nachfolgerfunktion S

$$\overline{\mathbf{2}}S\overline{\mathbf{3}} = \{\lambda s.\lambda z.s(s(z))\}\{\lambda wyx.y(wyx)\}\{\lambda a.\lambda b.a^3(b)\} \rightarrow$$
$$\rightarrow\{\lambda z.[\lambda w.\lambda y.\lambda x.y(wyx)]([\lambda w\lambda y.\lambda x.y(wyx)](z))\}\{\lambda a.\lambda b.a^3(b)\}$$
$$\rightarrow[\lambda w\lambda y.\lambda x.y(wyx)]([\lambda w.\lambda y\lambda.x.y(wyx)](\lambda a.\lambda b.a^3(b))) \equiv SS\overline{\mathbf{3}}$$

- $SS\overline{\mathbf{3}} \rightarrow S\overline{\mathbf{4}} \rightarrow \overline{\mathbf{5}}$

Multiplikation

- zwei Zahlen m und n multipliziert man durch $\lambda m.\lambda n.\lambda z.m(nz)$
- Produkt von 2 mit 2 ist dann $(\lambda m.\lambda n.\lambda z.m(nz))\overline{\mathbf{2}}\,\overline{\mathbf{2}}$
- reduziert zu $\lambda z.\overline{\mathbf{2}}(\overline{\mathbf{2}}z)$, weitere Reduktion ergibt $\overline{\mathbf{4}}$.

Logische Konstanten und Funktionen

- logische Konstanten $\mathbf{T} \equiv \lambda x\lambda y.x$ und $\mathbf{F} \equiv \lambda x\lambda y.y$
- logische Funktionen: $\wedge \equiv \lambda x\lambda y.xy\mathbf{F}$, $\vee \equiv \lambda x\lambda y.x(\lambda u\lambda v.u)y$ und $\neg \equiv \lambda x.x\mathbf{FT}$

Die Negationsfunktion angewendet auf $\mathbf{T}$ ist

$$(\lambda x.x\mathbf{FT})(\lambda a\lambda b.a) \equiv (\lambda x.x(\lambda c\lambda d.d)(\lambda e\lambda f.e))(\lambda a\lambda b.a)$$

was reduziert werden kann zu:

$$\mathbf{TFT} \equiv (\lambda a\lambda b.a)(\lambda c\lambda d.d)(\lambda e\lambda f.e) \rightarrow_\beta (\lambda c\lambda d.d) \equiv \mathbf{F}$$

Konditionale

- hilfreich bei Programmierung ist eine Test-auf-Null-Funktion f mit $f(0) = \mathbf{T}$ und $f(n) = \mathbf{F}$ für $n \neq 0$
- ein λ-Term für eine solche Funktion ist:

$$Z \equiv \lambda n.n\mathbf{F}\neg\mathbf{F}$$

- für jedes f ist nullfache Anwendung von f auf a gerade a

$$\overline{\mathbf{0}}fa \equiv (\lambda s.\lambda z.z)fa \equiv a$$

- außerdem $\mathbf{F}a \equiv \lambda y.y = I$ für beliebiges a

$$Z\overline{\mathbf{0}} \equiv (\lambda n.n\mathbf{F}\neg\mathbf{F})\overline{\mathbf{0}} = \overline{\mathbf{0}}\mathbf{F}\neg\mathbf{F} = \neg\mathbf{F} = \mathbf{T}$$

$$Z\overline{\mathbf{n}} \equiv (\lambda x.x\mathbf{F}\neg\mathbf{F})\overline{\mathbf{n}} \equiv \overline{\mathbf{n}}\mathbf{F}\neg\mathbf{F} = I\mathbf{F} = \mathbf{F}$$

Vorgängerfunktion P

- für den Vorgänger von $\overline{\mathbf{n}}$ konstruiert man das Paar $(\overline{\mathbf{n}}, \overline{\mathbf{n}-\mathbf{1}})$ und nimmt das zweite Element
- ein Paar (a, b) wird als $\lambda z.zab$ dargestellt
- $(\lambda z.zab)\mathbf{T} = \mathbf{T}ab = a$ und $(\lambda z.zab)\mathbf{F} = \mathbf{F}ab = b$
- λ-Ausdruck für Φ mit: $\Phi : (\overline{\mathbf{n}}, \overline{\mathbf{n}-\mathbf{1}}) \mapsto (\overline{\mathbf{n}+\mathbf{1}}, \overline{\mathbf{n}-\mathbf{1}})$

$$\Phi \equiv (\lambda p.\lambda z.z(S(p\mathbf{T}))(p\mathbf{T}))$$

- $p\mathbf{T}$ ergibt das erste Element des Paares p
- wende Φ n-mal auf das Paar $(\lambda z.z\overline{\mathbf{0}}\overline{\mathbf{0}})$ an, bilde zweite Projektion

$$P \equiv (\lambda n.n\Phi(\lambda z.z\overline{\mathbf{0}}\overline{\mathbf{0}})F)$$

- beachte: der Vorgänger von 0 ist 0.

Funktionen höherer Ordnung

- kein Unterschied zwischen einfachen Objekten z.B. Zahlen und komplexen Objekten wie Funktionen von Funktionen
- was sich als λ-Term formulieren lässt, kann durch andere λ-Terme manipuliert werden
- $Q := \lambda x. * \, x\, x$ ist Term für Quadrieren
- $P_8 := \lambda x.Q(Q(Q\, x))$ ist Term für 8. Potenz
- Term, für dreimalige Funktionsanwendung $T : \lambda f.(\lambda x.f(f(f\, x)))$, damit gilt $P_8 \equiv T\, Q$ und 5^8 ist $T\, Q\, 5$
- $T\, T$ angewendet auf eine Funktion f ergibt f^{27}
- Operatoren wie T heißen Funktionen höherer Ordnung

Iterationen und Rekursion

- wiederholte Funktionsanw. als Kombination von λ-Termen
- wollen Verhalten einer `FOR`-Schleife darstellen, bei der die Anzahl der Wiederholungen durch Zähler kontrolliert wird
- haben $Z \equiv \lambda n.n\mathbf{F}\neg\mathbf{F}$ mit $Z\overline{\mathbf{0}}xy = x$ und $Z\overline{\mathbf{n}}xy = y$ für $n \neq 0$, sowie Vorgänger P und Nachfolger S
- suchen $\mathbf{I}$ mit $\mathbf{I}\,\overline{\mathbf{n}}\, f\, x = f(f(f \ldots (f\, x) \ldots))$ und $\mathbf{I}\,\overline{\mathbf{0}}\, f\, x = x$
- probiere: $\mathbf{I} = \lambda n.\lambda f.\lambda x.Z\, n\, x(\mathbf{I}(P\, n)f(f\, x))$
- $\mathbf{I}$ steht sowohl rechts als auch links, keine echte Definition
- man kann $\mathbf{I}$ aber als Fixpunkt des rechten Terms auffassen

$$A := \lambda M.(\lambda n.\lambda f.\lambda x.Z\, n\, x(M(P\, n)f(f\, x))))$$

Für welches $\mathbf{I}$ ist $\mathbf{I} = A\,\mathbf{I}$? Wie findet man Fixpunkte von A?

Fixpunktkombinator

- suchen Fixpunkt für einen Term
- man kann λ-Terme $\mathbf{Y}$ angeben, die einen Fixpunkt für einen beliebigen Term angeben, d.h.

$$\exists Y.\forall M. \qquad Y\,M = M\,(Y\,M)$$

- mit diesem $\mathbf{Y}$ lösen wir $\mathbf{I} = A\,\mathbf{I}$ durch $\mathbf{I} := \mathbf{Y}\,A$

$$\mathbf{Y} := (\lambda y.(\lambda x.y(xx))(\lambda x.y(xx)))$$

$$\begin{aligned} \mathbf{Y}R &= (\lambda y.(\lambda x.y(xx))(\lambda x.y(xx)))R \\ &\to_\beta (\lambda x.R(xx))(\lambda x.R(xx)) \\ &\to_\beta R((\lambda x.R(xx))(\lambda x.R(xx))) \\ &\equiv R(\mathbf{Y}R) \end{aligned}$$

R wird ausgewertet durch rekursiven Aufruf von $\mathbf{Y}R$ als erstes Argument.

Fixpunktkombinator am Beispiel

Berechne

$$f(n) = \sum_{i=0}^{n} i = n + \sum_{i=0}^{n-1} i$$

Sei $R = (\lambda r \lambda n.Zn\overline{\mathbf{0}}(nS(r(Pn))))$ wende S n-mal an, falls $n \neq 0$)

$$\begin{aligned} YR\overline{\mathbf{3}} &= R(YR)\overline{\mathbf{3}} \to Z\overline{\mathbf{3}}\overline{\mathbf{0}}(\overline{\mathbf{3}}S(YR(P\overline{\mathbf{3}}))) \\ &\to \overline{\mathbf{3}}S(YR\overline{\mathbf{2}}) \text{ (weil } 3 \neq 0) \\ &\to \ldots \to \overline{\mathbf{3}}S\overline{\mathbf{2}}S\overline{\mathbf{1}}S\overline{\mathbf{0}} \equiv \overline{\mathbf{6}} \end{aligned}$$

Rekursion bricht ab, wenn Argument 0 wird.

λ-Kalkül als Berechnungsmodell

- es gibt eine Vielzahl von λ-Kalkülvarianten
- ein Kalkül heißt Turing-mächtig oder Turing-vollständig, wenn in ihm alle berechenbaren Funktionen auf $\mathbb{N}$ ausgedrückt werden können
- β-Regel ist wohldefiniert und kann von einem Programm ausgeführt werden
- λ-Kalkül angereichert mit Z, P, S und mit Konstanten für alle Zahlen ist Turing-mächtig
- reiner λ-Kalkül ohne Konstanten ist Turing-mächtig

Bonusmaterial

Vermeidung schlechter Terme durch Typisierung

- wir haben Verwendung für seltsame Terme wie $\mathbf{Y} := (\lambda y.(\lambda x.y(xx))(\lambda x.y(xx)))$
- niemand hindert uns *sin log* zu bilden, obwohl *sin* nur auf Zahlen angewendet werden sollte, Compiler einer vernünftigen Programmiersprache sollte ablehnen
- Typisieren der Terme: welche Arten von Argumenten akzeptiert ein Term, welche Art Ergebnis produziert er
- Beispiel $sin : \mathbb{R} \to \mathbb{R}$
- einfaches Typsystem $\tau = c \mid \tau \to \tau$ besteht aus Basistypen c und Funktionentypen $\tau \to \tau$

Wohlgetypte λ-Terme

Definition

Basisfall: *für jeden Typ σ und Variable x ist der Term $x : \sigma$ wohlgetypt und hat Typ σ*

Abstraktion: *für $M : \tau$ und Variable $x : \sigma$ ist $\lambda x : \sigma.M$ wohlgetypt und hat Typ $\sigma \to \tau$*

Applikation: *falls M wohlgetypt vom Typ $\sigma \to \tau$ und N wohlgetypt vom Typ σ, dann ist $M\,N$ wohlgetypt vom Typ τ*

Beispiele:

1. $\lambda x : \sigma.x : \sigma$ hat Typ $\sigma \to \sigma$
2. $\lambda x : \sigma.\lambda y : \tau.x : \sigma$ hat den Typ $\sigma \to (\tau \to \sigma)$
3. *sin log* ist nicht wohlgetypt, $M\,M$ läßt sich niemals typisieren, egal für welches M

Rückgewinnung der Turing-Mächtigkeit

Satz

Jeder wohlgetypte Term hat eine Normalform.

- $\mathbf{Y}$ ist also nicht wohlgetypt, gehört nicht zum einfach getypten λ-Kalkül
- einfach getypter λ-Kalkül ist nicht Turing-mächtig
- man kann ihn anreichern mit Fixpunkt-Kombinator-Konstanten
- System PCF (programming computable functions), eingeführt von Scott und Plotkin, ist Turing-mächtig.

System PCF

besteht aus λ-Termen für ein einfaches Typsystem mit Basistyp `int` und folgenden Konstanten:

1. $\overline{\mathbf{n}}$ vom Typ `int` für jede natürliche Zahl n
2. Konditional Z_σ vom Typ $\texttt{int} \to (\sigma \to (\sigma \to \sigma))$
3. P und S vom Typ $\texttt{int} \to \texttt{int}$ für Vorgänger- und Nachfolgerfunktion
4. je ein Y_σ vom Typ $(\sigma \to \sigma) \to \sigma$