

Zusammenfassung

Autor: Frieder H.

Inhaltsverzeichnis

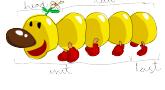
1 Haskell	1
1.1 Listen	1
1.2 Funktionsanwendungen	1
1.3 Lokale Namensbindung	1
1.4 Folds	2
1.5 Typen	2
1.6 Monaden	3
2 λ-Kalkül	4
2.1 untypisiertes λ-Kalkül	4
2.2 Church-Zahlen	4
2.3 Regelsysteme	5
2.4 Typsysteme	7
2.5 Typinferenz	8
3 Handschrift	8

1 Haskell

Linear Rekursiv: In jeden Definitionszweig kommt nur ein rekursiver Aufruf vor.
Endrekursiv: Linear rekursiv in jeden Zweig ist der rekursive Aufruf nicht in andere Aufrufe eingebettet.

1.1 Listen

`[] , x:xs`



`head [1,2,3] => 1, tail [1,2,3] => [2,3]`
`init [1,2,3] => [1,2], last [1,2,3] => 3`
`take n l -- erste n Elemente von l`
`drop n l -- l ohne erste n Elemente`

`takewhile bedingung liste`
`dropwhile bedingung liste`
`span b l == (takewhile b l, dropwhile b l)`

`null [1,2,3] => False`
`length`
`isIn`

`a ++ b`
`xs !! n -- Nte Listenelement`
`reverse xs`
`map f (x:xs) -- wendet f auf alle Listenelemente an`
`filter pred (x:xs) -- behalte alle Elemente, die Prädikat erfüllen`

1.2 Funktionsanwendungen

Funktionskomposition $f \circ g$: `comp f g = (\lambda x \rightarrow f(g x))` Infix: `f . g`

n-Fache Funktionsanwendung f^n : `iter f n`

Funktionstypen sind rechts-assoziativ, Funktionsanwendung ist links-assoziativ

`f x = y + x` Variable x Gebunden, Variable y frei.

1.3 Lokale Namensbindung

`energy m = let c = 299`
 `square = x = x * x`
 `in m * (square c)`

`energy m = m * (square c)`
 `where c = 299`
 `square x = x * x`

Einrückung hat semantische Bedeutung.
Bei Schachtelung: Inneres `let` bindet stärker.

1.4 Folds

```

foldr operator initial [] = initial
foldr operator initial (x:xs) = operator x (foldr operator initial xs)
foldr f z [x1, x2, ..., xn] == x1 `f` (x2 `f` ... (xn `f` z)...)
foldl op i [] i
foldl op i (x:xs) = foldl op (op i x) xs
foldl f z [x1, x2, ..., xn] == (...((z `f` x1) `f` x2) `f` ...) `f` xn

Beispiel: length list = foldr (\x n -> n + 1) 0 list

concat [xs, ys, zs] = xs ++ ys ++ zs concat == foldr app []

zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f xs ys = []
zip = zipWith (,) -- zip[1,2,3],[7,8,9] = [(1,7),(2,8),(3,9)]

```

Kurznotation Intervalle $[a..b] \Rightarrow^+ [a, a+1, a+2, \dots, b]$
List Comprehensions $[e \mid q_1, \dots, q_m]$, q_i Tests oder Generatoren der Form $p \leftarrow list$ mit Muster p und Listenausdruck $list$
Beispiel: squares n = $[x * x \mid x <- [0..n]]$
Im Muster p gebundene Variablen können in e und in q_i verwendet werden
Beispiel: evens n = $[x \mid \underbrace{x <- [0..n]}_{\text{Generator}}, \underbrace{x \bmod 2 == 0}_{\text{Tester}}]$
Unendliche Listen: odds = $1 : \text{map } (+2) \text{ odds}$

```

iterate f a = a : iterate f (f a)
odds = iterate (+2) 1
iterate f x !! 23 -- führt "Schleife" 23 mal aus

```

1.5 Typen

```

type neuerName = [Integer]
    = String
    = (a, b)
    = ...

```

Algebraische Datentypen

```

data Shape = Circle Double
           | Rectangle Double Double

```

Konstruktoren

```

Circle :: Double -> Shape
Rectangle :: Double -> Double -> Shape

```

Aufzählungstypen

```
data Season = Spring | Summer | Autumn | Winter
```

Polymorphe Datentypen (Optionale Werte)

```
data Maybe t = Nothing | Just t
data Either s t = Left s | Right t
data Matrix t = Dense [[t]] -- Liste von Zeilen
               | Sparse [(Integer, Integer t)] t -- Einträge (i, j, v) und Defaultwert
data Stack t = Empty | Stacked t (Stack t)
```

Polymorphe Funktion mit Typeinschränkung

```
qsort :: Ord t => [t] -> [t]           Instanzen von Ord implementieren
qsort [] = []
qsort (p:ps) = ...                         <=, <, >, >=, ...
```

Typklassen-Definitionen

```
instance Eq Bool where
    True == True = True
    True == False = False
    :
    :
```

Fehlende Implementierungen: Default implementiert

```
data shape = Circle Double
            | Rectangle Double Double
deriving Eq -- Keine eigene (==) Funktion mehr notwendig
```

Automatische Instanziierung auch für **Show,Ord,Enum**

1.6 Monaden

2 λ-Kalkül

2.1 untypisiertes λ -Kalkül

	Notation	Beispiele
Variablen	x	x
Abstraktion	$\lambda x.t$	$\lambda y.0$
Funktionsanwendung	t_1t_2	$f42$

Funktionsanwendung linksassoziativ, bindet stärker als Abstraktion

α -Äquivalenz	t_1 und t_t heißen α -äquivalent $t_1 \stackrel{\alpha}{=} t_2$, wenn t_1 in t_2 durch konsistente Umbenennung der λ -gebundenen Variablen überführt werden kann
η -Äquivalenz	Terme $\lambda x.f x$ und f heißen η -äquivalent $(\lambda x.f x \stackrel{\eta}{=} f)$ falls x nicht freie Variable vob f
Redex	Ein λ -Term der Form $(\lambda x.t_1) t_2$ heißt Redex
β -Reduktion	β -Reduktion entspricht der Ausführung der Funktionsanwendung auf einen Redex
	$(\lambda x.t_1)t_2 \Rightarrow t_1[x \mapsto t_2]$
Substitution	$t_1[x \mapsto t_2]$ erhält man aus den Term t_1 , wenn man alle freien Vorkommen von x durch t_2 ersetzt
Normalform	Ein Term, der nicht weiter reduziert werden kann
Volle β -Reduktion	Jeder Redex kann jederzeit reduziert werden
Normalreihenfolge	Immer der linkste äußerste Redex wird Reduziert

let $x = t_1$ *in* t_2 wird zu $(\lambda x.t_2) \ t_1$

2.2 Church-Zahlen

Eine (natürliche) Zahl drückt aus, wie oft die Funktion s angewendet wurde

	Nachfolgefunktion
$c_0 = \lambda s. \lambda z. z$	
$c_1 = \lambda s. \lambda z. s \ z$	$\text{succ} = \lambda n. \lambda s. \lambda z. s \ (n \ s \ z)$
$c_2 = \lambda s. \lambda z. s \ (s \ z)$	wobei n Church Zahl
⋮	
$c_n = \lambda s. \lambda z. s^n \ z$	

Addition	$plus = \lambda m. \lambda n. \lambda z. m\ s\ (n\ s\ z)$
Multiplikation	$times = \lambda m. \lambda n. \lambda s. n(m\ s)$
	$\stackrel{\eta}{=} \lambda m. \lambda n. \lambda s. \lambda z. n(m\ s)\ z$
Potenzieren	$exp = \lambda m. \lambda n. n\ m$
	$\stackrel{\eta}{=} \lambda m. \lambda n. \lambda s. \lambda z. n\ m\ s\ z$
$c_{true} = \lambda t. \lambda f. t$	$c_{false} = \lambda t. \lambda f. f$
$isZero = \lambda n. n\ (\lambda x. c_{false})\ c_{true}$	

Rekursionsoperator $Y = \lambda f. (\lambda x. f(x\ x))(\lambda x. f(x\ x))$

$$f(Y\ f) \stackrel{\eta}{=} Y\ f$$

$Y\ f$ ist Fixpunkt von f

Church-Rosser: Wenn $t \Rightarrow t_1$ und $t_1 \Rightarrow^* t_2$, dann gibt es t' mit $t_1 \Rightarrow t'$ und $t_2 \Rightarrow^* t'$

Call-by-name	Reduziere linken äußersten Redex aber nicht, falls von einem λ umgeben → Reduziere Argumente erst, wenn benötigt
Call-by-Value	Reduziere linken Redex, der nicht von einem λ umgeben und dessen Argument ein Wert → Argument vor Funktionsaufruf ist auszuwerten

2.3 Regelsysteme

Fregescher Schlussstrich

$$\frac{\varphi_1 \quad \varphi_2 \quad \varphi_3 \quad \dots \quad \varphi_n}{\varphi}$$

Jede Regel stellt Implikation dar, φ_i Voraussetzungen, φ Konklusion

Introduktionsregel

Konjunktion

$$\wedge I \frac{\psi \quad \varphi}{\psi \wedge \varphi}$$

Eliminationsregel

$$\wedge E_1 \frac{\psi \wedge \varphi}{\psi} \quad \wedge E_2 \frac{\psi \wedge \varphi}{\varphi}$$

All-Quantor

$$\forall I \frac{P(y) \text{ } y \text{ ist frei in } P}{\forall x. P(x)} \quad \forall E \frac{\forall x. P(x)}{P(y)}$$

Implikation

$$\begin{array}{c} \psi \\ \vdots^1 \\ \rightarrow I \frac{\varphi}{\psi \rightarrow \varphi} \end{array} \quad \text{MP} \frac{\psi \rightarrow \varphi \quad \psi}{\varphi}$$

Alternative Notation

$$\frac{\Gamma_1 \vdash \varphi_1 \quad \dots \quad \Gamma_n \vdash \varphi_n}{\vdash \varphi}$$

Regelsysteme alternative Notation

Introduktionsregeln

$$\wedge I \frac{\Gamma \vdash \varphi \quad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi}$$

$$\rightarrow I \frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \rightarrow \psi} \quad \text{ASSMI} \frac{}{\Gamma, \varphi \vdash \varphi}$$

$$\wedge I_1 \frac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \wedge \psi} \quad \dots$$

Eliminationsregeln

$$\wedge E_1 \frac{\Gamma \vdash \psi \wedge \psi}{\Gamma \vdash \varphi} \quad \wedge E_2 \frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \psi}$$

$$\text{MP} \frac{\Gamma \vdash \varphi \rightarrow \psi \quad \Gamma \vdash \varphi}{\Gamma \vdash \psi}$$

$$\text{VE} \frac{\Gamma \vdash \varphi \wedge \psi \quad \Gamma, \varphi \vdash \omega \quad \Gamma, \psi \vdash \omega}{\Gamma \vdash \omega}$$

¹Herleitung gemäß Regeln verbindet Implikation der Regeln mit prädikatenlogischer Implikation

2.4 Typsysteme

Funktionstypen rechtsassoziativ Typsystem $\Gamma \vdash t : \tau$ im Typkontext Γ hat Term t Typ τ .
 Γ ordnet freien Variablen x ihren Typ $\Gamma(x)$ zu.

$$\text{CONST} \frac{c \in \text{Const}}{\Gamma \vdash c : \tau_2}$$

$$\text{VAR} \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

$$\text{ABS} \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x. t : \tau_1 \longrightarrow \tau_2}$$

$$\text{APP} \frac{\Gamma \vdash t_1 : \tau_2 \longrightarrow \tau \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1 t_2 : \tau}$$

Typisierung von λ -Term t : Paar (Γ, τ) , sodass $\Gamma \vdash t : \tau$ herleitbar
 β -Reduktion: Substitution von x ($\lambda x. t_1$) $t_2 \Rightarrow t_1[x \mapsto t_2]$

Substitutionslemma

Wenn $\Gamma, x : \tau_2 \vdash t_1 : \tau_1$ und $\Gamma \vdash t_2 : \tau_2$ dann
 $\Gamma \vdash t_1[x \mapsto t_2] : \tau_1$

Typbehaltungstheorem

Wenn $\Gamma \vdash t : \tau$ und $t \Rightarrow t'$ dann $\Gamma \vdash t' : \tau$

Typschema

Typ der Gestalt $\forall \alpha_1. \forall \alpha_2. \dots \forall \alpha_n$ heißt Typschema.

Instanziierung eines Typschemas

Es bindet freie Typvariablen $\alpha_1 \dots \alpha_n$ in τ

Für nicht-Schema-Typen τ_2 ist der Typ $\tau[\alpha \mapsto \tau_2]$

eine Instanzierung vom Typschema $\forall \alpha. \tau$

Schreibweise $(\forall \alpha. \tau) \succeq \tau[\alpha \mapsto \tau_2]$

Beispiele

$\forall \alpha. \alpha \rightarrow \alpha \succeq \text{int} \rightarrow \text{int}$

$\forall \alpha. \alpha \rightarrow \alpha \succeq (\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int}) \quad \text{int} \succeq \text{int}$

$\alpha \rightarrow \alpha \not\succeq \text{int} \rightarrow \text{int} \quad \alpha \not\succeq \text{bool}$

$\forall \alpha. \alpha \rightarrow \alpha \not\succeq \text{bool}$

2.5 Typinferenz

$$\text{VAR} \frac{\Gamma(x) = \sigma \quad \sigma \succeq \iota}{\Gamma \vdash x : \tau}$$

Constraint: $\{\iota = \tau\}$

$$\text{APP} \frac{\Gamma \vdash f : \xi \quad \Gamma \vdash x : \varphi}{\Gamma \vdash f x : \alpha}$$

Constraint: $\{\xi = \varphi \rightarrow \alpha\}$

$$\text{ABS} \frac{\Gamma, p : \pi \vdash b : \beta}{\Gamma \vdash \lambda p. b : \alpha}$$

Constraint: $\{\alpha = \pi \rightarrow \beta\}$

$$\text{LET} \frac{\Gamma \vdash y : \pi \quad \Gamma' \vdash b : \beta}{\Gamma \vdash \text{let } x = y \text{ in } b : \tau}$$

Constraints: Siehe unten

$ta(\tau, \Gamma)$ bindet alle in Γ freien Typvariablen mit einem \forall in τ

Bsp.: $ta(\alpha \rightarrow \beta, x : \beta, y : \delta) = \forall \alpha. \alpha \rightarrow \beta$

$$\text{LET} \frac{\Gamma \vdash y : \pi \quad \Gamma' \vdash b : \beta}{\Gamma \vdash \text{let } x = y \text{ in } b : \tau}$$

Constraints: Finde Unifikator σ_{LET} und allg. Typ π für y

1. Sei C_0 die bisherige Constraintmenge inklusive $\{\tau = \beta\}$
2. Sammle Constraints aus linken Teilbaum in C_{LET}
3. Berechne mgu σ_{LET} von C_{LET}
4. Berechne $\Gamma' := \sigma_{\text{LET}}(\Gamma), x : ta(\sigma_{\text{LET}}(\pi), \sigma_{\text{LET}}(\Gamma))$
5. Benutze Γ' in rechten Teilbaum, sammle Constraints in C_1
6. Ergebnisconstraints sind $C_0 \cup C'_{\text{LET}} \cup C_1$
mit $C'_{\text{LET}} := \{\alpha_i = \sigma_{\text{LET}}(\alpha_i) | \sigma_{\text{LET}} \text{ definiert für } \alpha_i\}$

Typen sind Rechstassoziativ also wenn suche nach mgu und

$$\alpha_1 = \alpha_3 \rightarrow \alpha_4 \rightarrow \alpha_5 = (\alpha_3 \rightarrow (\alpha_4 \rightarrow \alpha_5))$$

$$\alpha_2 = \alpha_6 \rightarrow \alpha_7$$

dann

$$\alpha_6 \triangleleft \alpha_3$$

$$\alpha_7 \triangleleft \alpha_4 \rightarrow \alpha_5$$

3 Handschrift

Ziel dieses Dokumentes war es die handschriftliche Zusammenfassung abzutippen. Leider war das wesentlich schwieriger und zeitaufwändiger als gedacht.

Deswegen „for completeness Sake“ im folgenden die Handschriftliche version auf der diese Zusammenfassung basiert

Propa Zusammenfassung

Haskell: ~~0.99~~

Linear Rekursiv: In jeder Definition zweig kommt nur ein rekursiver Aufruf
Endlichrekursiv: linear rekursiv in jeder Zweig, z.B. der rekursive Aufruf ist nicht in andere Aufrufe eingebettet.

Listen: $[]$, $x:xs$ head $[1,2,3] \Rightarrow 1$, tail $[1,2,3] \Rightarrow [2,3]$
 $null [1,2,3] \Rightarrow \text{False}$, length, isIn
 $att 6$, take n erste n Elemente von l
 $drop n$ letzte n Elemente von l
 $init [1,2,3] = [1,2]$, last $[1,2,3] = 3$
 $map f (x:xs)$ wendet f auf alle Listenelemente an.

Filter pred ($x:xs$) Behalte Elemente, die Prädikat erfüllen.

Funktionskomposition Log: comp $f g = ((x \rightarrow f(g(x)))$ h.h.k: $f \circ g$
 n -fache Funktionsanwendung f^n iter $f n$

Funktionstypen sind rechts-assoziativ, Funktionsanwendung ist links-assoziativ
 $f \circ g = g \circ f$ x vor x gebunden, y frei.

Lokale Namensbindung:

energy m = let c = 299

square x = $x * x$
 in $m * (\text{square } c)$

energy m = m * (square c)

where $c = 299$

square x = $x * x$

Einrückung hat semantische Bedeutung.

Bei Schachtelung: Innenes let bindet stärker

Folds:

- foldr operator initial $[] = \text{initial}$
- foldr operator initial $(x:xs) = \text{operator} x (\text{foldr operator initial } xs)$
- foldr $f = [x_1, x_2, \dots, x_n] = x_1 'f' (x_2 'f' \dots (x_n 'f' z))$
- foldl op i $[] = i$
- foldl op i $(x:xs) = \text{foldl op} (\text{op } i \text{ } x) xs$
- foldl $f = [x_1, x_2, \dots, x_n] = (\dots ((z 'f' x_1) 'f' x_2) f \dots) 'f' x_n$

Beispiel: $\text{length } \underline{\text{List}} = \text{foldr } (\lambda n \rightarrow n+1) 0$

concat $[xs, ys, zs] = xs ++ ys ++ zs$ concat = foldr app $[]$

zipWith $f (x:xs) (y:ys) = f x y : \text{zipWith } f xs ys$
 $\text{zipWith } f xs ys = []$

zip = zipWith (,) $\text{zip } [1,2,3] [9,8,7] = [(1,9), (2,8), (3,7)]$
 takeWhile elementKdBy $\text{takeWhile } b l = \text{takeWhile } b (l, \text{rest})$
 horizontation Intervalle $[a .. b] \rightarrow [a, a+1, a+2, \dots, b]$
 $[a ..]$ unendl. intervall, startFirst a

List Comprehensions $[e | q_1, \dots, q_m], q_i$ Tests oder Generatoren
 des Form $p \in \text{list}$
 mit Klammer p und Liste ausdrückt list

Paarung eines $n = \{x \mid x \in [0, n], x \bmod 2 = 0\}$

Kapitel 2

Endliche Listen: $\text{odds} = \underbrace{n : \text{map} (+2)}_{\text{gerade}} \text{ odds}$

iterate $f a = a : \text{iterate } f (f a)$

$\text{odds} = \text{iterate } (+2) 1$

iterate $f x !! 23$ führt "Schleife" 23 Mal aus repeat $\Rightarrow \text{iterate} (\lambda x \rightarrow x)$

type readName = [$\begin{array}{l} \text{Integers} \\ \text{String} \\ \text{Double} \end{array}\right]$

• Algebraische Datentypen: ~~durch Autzählnat Konstrukte~~

data Shape = Circle Double

Rectangle (x: Double) Double

Autzählnat Typen: Rectangle Double Double

Circle : Double \rightarrow Shape

data Season = Spring | Summer | Autumn | Winter

• Polymorphe Datentypen (Optionale Werte)

data Maybe t = Nothing | Just t

data Either s t = Left s | Right t

data Matrix t = Dense [[t]] - Liste von Zeilen
ISparse [(Integers, Integers, t)] t - Einträge (i, j, v) und
Obertrindest.

data Stack t = Empty | Stacked t (Stack t)

Polymorphe Typen Funktion mit Typengeschränkung

qsort :: Ord t \Rightarrow [t] \rightarrow [t] instanzieren t von Ord implementieren
 $\langle =, <, >, \geq, \ldots \rangle$

qsort [] = []
qsort (x: xs) = ...

Fayhlassen-Deklarationen

instance Eq Bool where

True == True = True

False == False = False

Fehlende Implementierungen: Default implementiert

data Shape = Circle Double

| Rectangle Double Double

delegating Eq keine eigene ($==$) Funktion mehr notwendig,
Automatische Instanziierung auch für Show, Ord, Enum

Moranden

untypisierte λ -Kalkül

	Notation	Bespreche
Variablen	x	x
Abspracht	$\lambda x.t$	$\lambda y.0$
Funktionsanwendung	$t_1 t_2$	$\lambda f. \lambda x. f y$, $f y$

Funktionsanwendung linksassoziativ, bindet stärker als Abstraktion

α -Äquivalenz: t_1 und t_2 heißen ab α -äquivalent $t_1 \alpha \equiv t_2$ wenn t_1 in t_2 durch konstante Umformung der λ -gebundenen Variablen überführt werden kann.

η -Äquivalenz: Forme $\lambda x. t x$ und t heißen η -Äquivalent, $(\lambda x. x t \equiv t)$ falls x nicht freie Variable von t ist

Redex: Ein λ -Term der Form $(\lambda x. t_1) t_2$ heißt Redex

β -Reduktion: β -Reduktion entspricht der Ausführung des Funktionsanwendungs auf einen Redex

$$(\lambda x. t_1) t_2 \Rightarrow t_1[x \mapsto t_2]$$

Substitution: $t_1[x \mapsto t_2]$ erhält man aus dem Term t_1 , wenn man alle freien Vorkommen von x durch t_2 ersetzt.

Nachstform: Ein Term, der nicht weiter reduziert werden kann

Volle β -Reduktion: Jeder Redex kann gezielt reduziert werden

Normalbegriff: immer das linkeste auftrete Redex wird reduziert

let $x = t_1$ in t_2 wird zu $(\lambda x. t_2) t_1$

Church-Zahlen: Eine (natürliche) Zahl deutet aus, wie oft die Funktion s angewendet wurde

$$\begin{aligned} c_0 &= \lambda s. \lambda z. z \\ c_1 &= \lambda s. \lambda z. s z \\ c_2 &= \lambda s. \lambda z. s(s z) \\ &\vdots \\ c_n &= \lambda s. \lambda z. s^n z \end{aligned}$$

Subtraktion: sub

$$\text{Addition: plus} = \lambda m. \lambda n. \lambda s. \lambda z. m s (n s z)$$

$$\text{Multiplikation: times} = \lambda m. \lambda n. \lambda s. \lambda (m s)$$

$$\qquad \qquad \qquad \# \lambda m. \lambda n. \lambda s. \lambda z. m (n s z)$$

$$\text{Potenzieren: exp} = \lambda m. \lambda n. \lambda m^m$$

$$\qquad \qquad \qquad \# \lambda m. \lambda n. \lambda s. \lambda z. n^m s z$$

$$\text{true} = \lambda t. \lambda f. t \qquad \text{false} = \lambda t. \lambda f. f$$

$$\text{isZero} = \lambda n. \lambda (x. \text{false}) (x. \text{true})$$

Nachfolgerfunktion:

$$\text{succ} = \lambda n. \lambda s. \lambda z. s (n s z)$$

wobei n Church-Zahl

Rekursionsoperator

$$\eta y = \lambda t. (\lambda x. f(x x))(\lambda x. t(x x))$$

$$f(yt) \stackrel{\beta}{=} yt$$

yt ist Fixpunkt von f .

Church-Rössel

Wenn $t \leq t_1$ und $t \leq t_2$
dann gibt es t' mit $t_1 \leq t'$ und $t_2 \leq t'$

Call-by-name: Reduzierung ~~aller~~ linken äußersten Redex
wenn nicht falls von einer λ umgeben
 \rightarrow Reduzierbare Argumente oft nur benötigt.

Call-by-value Reduzierung linken Redex der nicht von einem λ umgeben
und dessen Argument ein Wert ist
 \rightarrow Argumente von Funktionsaufrufen auswerten.

Regelsysteme:

Fregesches Schlußstück:

$$\frac{\varphi_1 \quad \varphi_2 \quad \dots \quad \varphi_n}{\varphi} \quad \text{Jede Regel stellt} \\ \text{Implikation dar,} \\ \varphi_i \text{ Voraussetzung,} \\ \varphi \text{ Konklusion}$$

Introduktionsregel

$$\text{Konjunktion } \frac{\varphi_1 \quad \varphi}{\varphi_1 \wedge \varphi}$$

$$\text{NE}_1 \frac{\varphi_1 \wedge \varphi}{\varphi} \quad \text{NE}_2 \frac{\varphi_2 \wedge \varphi}{\varphi}$$

$$\text{All-Quantor } \forall x \frac{P(y)}{P(x)} \quad y \text{ ist frei in } P$$

$$\text{Ex-Quantor } \exists x \frac{P(x)}{P(y)}$$

$$\begin{array}{l} \text{Hilfsleitung gen. } \varphi \\ \text{Regeln verbindet } \varphi \\ \text{Implikation ist Regeln mit Rohbildungswelt } \varphi \\ \text{Implikation } \rightarrow I \quad \varphi \rightarrow \varphi \end{array}$$

$$\text{MP } \frac{\varphi \rightarrow \varphi \quad \varphi}{\varphi}$$

Alternative Notation

$$\frac{\Gamma_1 + \varphi_1 \quad \dots \quad \Gamma_n + \varphi_n}{\Gamma + \varphi}$$

Γ : Liste von Aussagen, gen. Kontext
 $\Gamma + \varphi$: φ unter Annahmen Γ herleitbar

\rightarrow

→ Regelsysteme alternative Notation:

Introduktionsregeln

$$\lambda I \frac{\Gamma \vdash \psi}{\Gamma \vdash \psi \wedge \psi}$$

$$\neg I \frac{\Gamma, \psi \vdash \psi}{\Gamma \vdash \psi \rightarrow \psi} \text{ Assm } \frac{}{\Gamma, \psi \vdash \psi}$$

$$\vee I_1 \frac{\Gamma \vdash \psi}{\Gamma \vdash \psi \vee \psi}$$

...

Eliminationsregeln

$$\wedge E_1 \frac{\Gamma \vdash \psi_1 \psi}{\Gamma \vdash \psi} \quad \wedge E_2 \frac{\Gamma \vdash \psi_2}{\Gamma \vdash \psi}$$

$$MP. \quad \frac{\Gamma \vdash \psi \rightarrow \psi}{\Gamma \vdash \psi}$$

$$\vee E \frac{\Gamma \vdash \psi \vee \psi \quad \Gamma, \psi \vdash w \quad \Gamma, \psi \vdash w}{\Gamma \vdash w}$$

Type systeme

Funktionsarten einstassoziativ

Type system $\Gamma \vdash t : \tau$ im Typkontext Γ hat Term t den Typ τ
falls freien Variablen x ihren Typ $\Gamma(x)$ zul.

$$\text{const } \frac{c \in \text{Const}}{\Gamma \vdash c : \tau_c}$$

$$\text{var } \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

$$\text{Abs } \frac{\Gamma, x : \tau_1, t : \tau_2}{\Gamma \vdash \lambda x. t : \tau_1 \rightarrow \tau_2}$$

$$\text{App } \frac{\Gamma \vdash t_1 : \tau_1 \rightarrow \tau \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1 t_2 : \tau}$$

Typisierung von λ -Term t : Paar (Γ, t) , sodass $\Gamma \vdash t : \tau$ lehrbar
A-Reduktion: Substitution von x

$$(x : \tau_1 \tau_2 \Rightarrow \tau_1[\tau_2 \mapsto \tau_2])$$

-Substitutionslemma: Wenn $\Gamma, x : \tau_2 \vdash t : \tau_1$ und $\Gamma \vdash \tau_2 : \tau_1$ dann $\Gamma \vdash t[\tau_2 \mapsto \tau_1]$

-Typ erhaltungstheorem: Wenn $\Gamma \vdash t : \tau$ und $t \Rightarrow t'$ dann $\Gamma \vdash t' : \tau$

Type schema: Typ der Gestalt $\forall x_1. \forall x_2. \dots \forall x_n$ heißt Type schema
Es bindet freie Typvariablen $\alpha_1, \dots, \alpha_n$ in τ

Instanzierung eines Typeschemas: Für nicht-Schema-Typen τ_2 ist der
Typ $\forall x. \alpha \rightarrow \tau_2$ eine Instanzierung vom Typeschema $\forall x. \alpha$
Schreibweise $(\forall x. \alpha) \Sigma \tau_2$

Beispiele: $\forall x. \alpha \rightarrow \alpha \Sigma \text{int} \rightarrow \text{int}$ Aber: $\alpha \rightarrow \alpha \not\models \text{int} \rightarrow \text{int}$

$$\alpha. \alpha \rightarrow \alpha \Sigma (\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int})$$

$$\text{int} \Sigma \text{int}$$

$$\alpha \not\models \text{Bool}$$

$$\forall x. \alpha \rightarrow \alpha \not\models \text{Bool}$$

Angepasste Regeln

$$\forall \text{Var} \frac{\Gamma(x) = \tau' \quad \tau' \models \tau}{\Gamma \vdash x : \tau}$$

$\Gamma, x : \tau_1 \vdash t : \tau_2$ σ_1 kein Type schema

$$\text{Abs: } \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x. t : \tau_1 \rightarrow \tau_2}$$

Constant & App Regeln bleiben gleich

λ -gebundene Bezeichner werden polymorph

Prop 6

Typabstraktion: Das Typschemata $\tau_0(\bar{c}, \bar{\Gamma}) = \text{Var}_1 \cdot \text{Var}_2 \cdots \text{Var}_n \cdot \bar{\Gamma}$ heißt Typabstraktion von $\bar{\Gamma}$ relativ zu $\bar{\Gamma}$, wobei $\bar{c} \in FV(\bar{\Gamma}) \setminus FV(\bar{\Gamma})$
 \Rightarrow Verhindern Abstraktion von globalen Typvariablen im Schema

$$\Gamma \vdash t_1 : \bar{\Gamma}_1 \quad \Gamma, x : \tau_0(\bar{c}, \bar{\Gamma}) \vdash t_2 : \bar{\Gamma}_2$$

LET :

$$\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : \bar{\Gamma}_2$$

Logische Programmierung

Haller Seite

Terme: Atome: hang, inge beginnen mit Kleinbuchstaben, Komplexe: Hochzeit
 Zahlen: $3, 4, 5$
 Variablen: $X, -X$ beginnen mit Großbuchstaben oder Unterstrich.
 Termlisten: $3, 4, 5, X, \text{hang}$
 Zusammengesetzt hat sieht ($\text{Kritz}, \text{hoch}$), sieht (tritex, X)

Atom am Anfang eines zusammengesetzten Terms: Funktionsvordefinierte Funktionen folieren in Infixnotation ($3 < 4$)

Atome und variablenfreie Terme stehen für sich selbst

Variablen sind Platzhalter für unbekannte, Terme andin Regelwerk
 Fakten implizit universell quantifiziert.

Abfrage durch Projektion? eingeschränkt und mit Punkt beendet.
 Erkennbar, dass nicht erfüllbar.

Abfrage Abfrage mit Variable: versucht Abfrageheraus mit Daten durch zu untersuchen. Konjunktion von Teilzielen getrennt durch Komma

Klammer in Abfragen und Regeln: Logisches \wedge (anders als in Prolog)

Teilziel erstellt von links nach rechts \Rightarrow passende Instantienungen
 Nach Erstellung eines Teilziels: nächstes Teilziel ist instantiieren.

Regeln: term :- termlist. :- Als "Weg" zu legen.
 Wenn term ist, dann term

Vorabellen in Regelkopf implizit universell quantifiziert. existentiell
 Variablen die ausschließlich in Regelkörpfer vorhandenen implizit universell quantifiziert

Gruppe von Fakten/Regeln mit gleichen Funktionsnamen und gleicher Argumentzahl in Regelkopf heißt abgezweigt allein Sichtbar

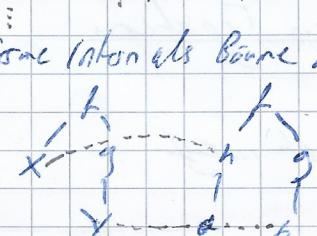
Beispiel:

grandparent(X, Y) :- parent(X, Z), parent(Z, Y). $\frac{\text{Z}}$ ist $\text{parent}(X, Z)$ und $\text{parent}(Z, Y)$

parent(X, Y) :- mother(X, Y).

parent(X, Y) :- father(X, Y).

mother(inge, emi).



Umstitution: Term unten als Börne dargestellt:

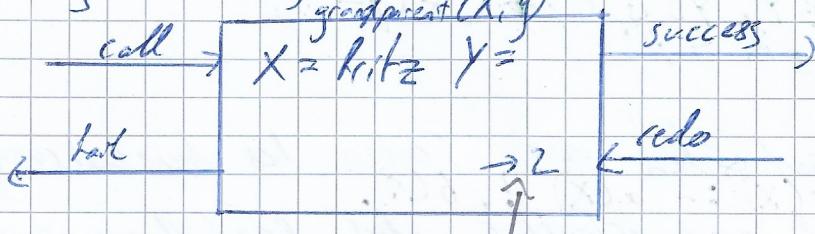
$$f(x, g(y)) = f(h(a), g(b)) \text{ mit } x \mapsto h(a) \\ y \mapsto b$$

Prägn 7

$f(x, y)$ und $g(x, y)$ nicht unifizierbar

$f(x, y)$ und $f(f(x, y), z)$ nicht unifizierbar
 $(x = f(x, y))$ durch keinen erfüllenden Term
 für x unifizierbar

Bachtracing: Visualisierung des Ausführungsbaums. Jedes Teilziel des Baums



Chance Pointe nächste zu probierende Regel bei Rechtsabgang versucht.

Im Beispiel: Regel 2 für grandparent(X, Y) erzielt nicht

Informativer Algorithmus auf Slide 24.1

Listen

$[X | Y]$: erstes Listenelement X cons Rest der Liste Y .

$[Z_1, Z_2, \dots, Z_n] \equiv [Z_1 | [Z_2 | [\dots | [Z_n | []] \dots]]]$

Y muss nicht instanziert sein \Rightarrow Listen können (anders als in Haskell) von vorne aufgebaut werden

member($X, [X|R]$).

member($X, [Y|R]$) :- member(X, R).

delete($[L_1, X, L_2]$) \forall Delibet X from L_1 , result in L_2

append($[], L, L$).

append($[X|R], L, [X|T]$) :- append(R, L, T). Wenn die Konkatenation von Rand R für Liste T ergibt, dann ergibt die Konkatenation von $[X|R]$ und L die Liste $[X|T]$.

reverse(X, Y) :- rev1($X, [], Y$).

rev1($[], Y, Y$).

rev1($[X|R], A, Y$) :- rev1($R, [X|A], Y$).

Auswertung arithmetischer Ausdrücke: Explizit per Teilziel 13:

X ist $B * (+)$ Variablen im Rechten müssen instanziert sein.

sum(A, B, C, S) :- S ist $A+B+C$. Nur vorwärts anwendbar:
 A, B, C müssen instanziert sein

Arithmetische Ausdrücke einifizieren nicht mit Menschen

Richtig

even 0.

even(X) :- X > 0, X1 is X-1, odd(X1).

odd(1).

odd(X) :- X > 1, X1 is X-1, even(X1)

Falsch

even(0)

even(X) :- X > 0, odd(X-1)

:

Generate and Test

erzeugen Lösungskandidaten und welche danach getestet werden.

nat(0).

nat(X) :- nat(Y), X is Y+1.

} Unendlich oft erstellbar.

sqrt(X, Y) :- nat(Y),

$\begin{cases} Y_2 \text{ ist } Y * Y, \\ Y_3 \text{ ist } (Y+1) * (Y+1), \\ Y_2 = < X, X < Y_3 \end{cases}$

Cut: ! Abscheiden von Teilen des Ausführungsbaums

Beispiel: p(X) :- a(X) !, b(X).

Als Testziel immer erstellbar, aber Regeldurchgänges versucht lässt alle Testziele links vom Cut sofort fall zu schlagen.

Blauer Cut beeinflusst weder Programmablaufzeit noch -verhalten
Grüner Cut beeinflusst Programmablaufzeit, aber nicht-verhalten.
Roter Cut beeinflusst Programmverhalten.

Sei B' Prädikat, dass genau dann erstellbar ist, wenn B nicht erstellbar ist
Dann ersetze ~~B~~ durch

A(X) :- B(X), C(X),
A(X) :- B'(X), D(X).

A(X) :- B(X), !, C(X).
A(X) :- D(X).

not(X) :- call(X), !, fail.

not(X).

atom(X) ist X mit einem Atom instanziiert?
atomic(X) ist X mit einem Atom oder einer Zahl instanziiert?

Dictionary Instanziierungen (N,A) von Wortsequenz-"Variablen"

D = [(1, Eich, bin), (2, Eich, denke)] L =]

Lookup(N,D,A) Instantiiert, D partiell instantiiert, A uninstantiiert
Definition Folie 281

X = Y unifiziert X mit Y, schlägt fehl, falls unmöglich
X = Z = Y stolzgleich, falls X und Y bereits unifiziert wurden.
X = := Y gleichsamer Vergleich, X und Y müssen instanziiert sein
X = .. L Konstruktion/Zerlegung von X aus/in Liste L

Freeze: Tests sollen automatisch so früh wie möglich ausgeführt werden

freezeall([X1, X2, ..., XT])

Wendet Testziel T nicht aus, wenn alle X1, X2, ... instanziiert sind.

Anwendungsschema: solve(L) :- freezeall(CL, best(CL)), generate(CL).

CL Listen von Variablen L

Substitution und Resolution

Gegeben: Menge C von Gleichungen über Terme, etwa

Typhusne $\Sigma = \underbrace{\text{BasisTyp}}_{\text{int. bdd}} \mid \underbrace{\text{Var}}_{\alpha_1, \alpha_2, \beta, \dots} \mid \underbrace{\text{Ex} \rightarrow \text{Ex}}_{\Sigma} \mid \Sigma$
z-stelliger Typhonstruktur

Prologterme $\Theta = \underbrace{\text{Atom}}_{\text{f. g. h.o.s., ...}} \mid \underbrace{\text{Var}}_{x, y, z, \dots} \mid \underbrace{\text{Atom}(\Theta_1, \dots, \Theta_n)}_{n\text{-stellige Funktion}}$

Substitution, die alle Gleichungen erfüllt: Unifikator
Substitution σ unifiziert Gleichung $\theta = \theta'$, falls $\sigma\theta = \sigma\theta'$.
 σ antrifft, falls $\theta \in C$ gilt & unifiziert θ .

Allgemeinstes Unifikator (most general unifier mgu)
 σ mgu, falls σ Unifikator $\forall \exists$ Substitution d. $\gamma = \delta \circ \sigma$

Unifikationsalgorithmus auf slide 299

Robinson Unifikator, Paterson-Weg von Unifikator \Rightarrow slide 302

Resolutionsregel

$(\alpha_1, \alpha_2, \dots, \alpha_n) \vdash \Gamma$ Terme; $\alpha := \alpha_1, \dots, \alpha_n$ eine Regel; σ mgu von α und Γ
 $\sigma(\alpha_1), \dots, \sigma(\alpha_n), \sigma(\Gamma_1), \dots, \sigma(\Gamma_m)$

Spezialfall u Fakt "h = c"

Start: Liste der α sind die Teile eines Abfrage
Falls die Liste leer ist, ist die Abfrage erfüllt.
Falls keine Regel mehr anwendbar ist, schlägt die Abfrage fehl.

Effiziente Resolutionsregel

$(\alpha_1, \alpha_2, \dots, \alpha_n) \vdash \Gamma$ Terme plus Substitution; $\alpha := \alpha_1, \dots, \alpha_n$ eine Regel
 σ mgu von α und $\gamma(\Gamma)$
 $(\alpha_1, \dots, \alpha_n, \beta_1, \dots, \beta_m \vdash \Gamma \sigma \gamma)$

- γ ist die Substitution, die bei den vorherigen Resolutions schritten berechnet wurde
- Wird die Liste der Teile leer (\emptyset), wird γ ausgegeben.

Resolutionprinzip

Sei P ein logisches Programm (i.e. eine Liste von Fakten und Regeln). Sei $\alpha_1, \dots, \alpha_n$ eine Liste von Zielen (Termen)

1. Die Schreibweise $P \models \alpha_1, \dots, \alpha_n$ bedeutet: Die Ziele $\alpha_1, \dots, \alpha_n$ legen sich mittels Resolution abarbeiten (mittels bestehender Fakten u. Regeln von P in die leere Zielliste verwandeln)

2. Die Schreibweise $P \models \alpha_1, \dots, \alpha_n$ bedeutet: Die Zielliste $\alpha_1, \dots, \alpha_n$ ist die logische Konsequenz aus den Fakten und Regeln des Programms P .

Korrektheit und Vollständigkeit: slide 309 u. 310.

Typinferenz: λ -Kalkül

Pagina 10

Vorgehen: Form (Γ, t) Problem: Finde Lösung (σ, τ)

Vorgehen:

- ① Erstelle Herleitungsbaum anhand symbolisches Skriptur und Typisierungregeln, Verwende zunächst überall freie Typvariable α_i
- ② Extrahiere Gleichungssystem C für die α_i gemäß Regeln
- ③ Bestimme allgemeinsten Abschluß σ_1 des Gleichungssystems
Lösung: $(\sigma, \sigma C \sigma_1)$ wobei σ_1 die für t passende Typvariable

Algorithmenalgorithmus: init $g(C) =$

```
if  $C == \emptyset$  then  $C'$ 
else let  $\xi$   $\in$   $C$   $\Rightarrow$   $\sigma_i \in C'$  =  $C$  in
    if  $\sigma_i == \sigma_1$  then var $g(\sigma_i)$ 
    else if  $\sigma_i == \alpha$  and  $\alpha \notin FV(C)$  then unity $(C[\alpha \mapsto \sigma_i])$ 
    else if  $\sigma_i == \alpha$  and  $\alpha \in FV(C)$  then unity $(C[\alpha \mapsto \sigma_i])$ 
    else if  $\sigma_i == (\sigma_1' \rightarrow \sigma_2')$  and  $\sigma_2' == (\sigma_2 \rightarrow \sigma_3')$ 
        then unity $(C' \cup g(\sigma_1' \sigma_2' \sigma_3'))$ 
    else fail
```

Typinferenz in Prolog (s. S. 331) / Let-Polytypismus in Prolog S. 310

Typinferenz für Let

- clarifizierbar σ_{let} für Let aus linken Teilbaum bestimmen
- Weiter im rechten Baum mit $\Gamma' \vdash \sigma_{let}(\Gamma, t, \sigma_{let}) : \text{bal}(\sigma_{let}, \sigma_{let})$
- Vereinfache $\sigma_{let} = \{\alpha_i == \sigma_{let}(\alpha_i)\}$ (Let definiert nur α_i) erstellt Constraints für restliche Typinferenz.

Allgemeine Vorgehensweise:

1. Sei C_0 die leere Constraintmenge initial. ($\{\alpha_i = \alpha_j\}$)

2. Sammle Constraints aus linkem Teilbaum in C_{let}

3. Berechne den neuen σ_{let} von C_{let}

4. Berechne $\Gamma' \vdash \sigma_{let}(\Gamma), \alpha : \text{bal}(\sigma_{let}, \sigma_{let})$

5. Berechne Γ' im rechten Teilbaum, sammle Constraints in $C_{\neg let}$

6. Ergebnisconstraints sind $C_0 \cup C_{let} \cup C_{\neg let}$ mit

$C_{let} := \{ \alpha_i == \sigma_{let}(\alpha_i) \mid \sigma_{let} \text{ definiert f. } \alpha_i \}$

Parallel Programming

Flynn's Taxonomy

1. SISD Single Instruction x Single Data von Neumann architecture, e.g. single, uniform memory access or as if
2. SIMD Single Instruction x Multiple Data one instruction is applied to homogeneous data eg. vector process
3. MIMD Multi Instruction x Multiple Data different processors operate on different data e.g. today's multiprocessor
4. MISD Multiple Instruction x Single Data on the same data e.g. pipeline

Amdahl's Law

Proprietary

$$S(n) = \frac{T(1)}{T(n)}$$

Possible Speedup of an algorithm execution on n processors. $T(x)$ is processing time it processed by x processors.

Maximal speedup with parallel processing, parallelizable percentage of the program

$$S(n) = \frac{1}{(1-p) + \frac{p}{n}}$$

Message Passing Interface (MPI)

Follows SIMD (single instruction, multiple data) model
The same program is started on n nodes.

Communicators

• Befehl • Beispiele mit
Schlagwort

- Processors communicate via so-called communicators
 - A group of processes, that can communicate with each other
 - MPI_COMM_WORLD** is the default communicator, i.e. the collection of all processes
 - Needs to be provided to almost every MPI command as a parameter
- Within a communicator with N processes, each process is identified by its individual rank $R; (0 \leq R < N)$
- MPI_Comm_size(MPI_COMM_WORLD, &my_rank)** retrieves the number of the each processing node (rank) of the program is running.

MPI_Comm_size(MPI_COMM_WORLD, &size) total number of processes in a communicator
Both return int, indicating success

MPI_Init(&argc, &args) initialize MPI
MPI_Finalize() clean up

MPI_BARRIER(MPI_Comm comm) blocks until all processes have called it

int MPI_Send(void *buffer, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

buffer the initial memory address of the senders buffer
count number of elements that will be send
datatype type of the buffers elements
tag context of the message (e.g. a conversion ID)
comm communicator of the process group

Blocking and asynchronous. blocks until the message buffer is released at **MPI_Probe**.

int MPI_Recv(void *buffer, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status &status)

source and tag need to match a send operation, wildcards possible
MPI_ANY_SOURCE, MPI_ANY_TAG

status, MPI_SOURCE and status, MPI_TAG contains source and tag
- count: fewer datatype elements can be received. There would be several **MPI_PROBE** value can be used, to receive message about message length

- Blocking and asynchronous. blocks until the message is received in the buffer completely

`int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)`
 Can be used to dynamically get the size of the message

Communication modes for send operations

<code>MPI_Send</code>	Synchronous	No bubbles, synchronization (both sides wait for each other)
<code>MPI_Broadcast</code>		Explicit buffering, no synchronization (no wait for each other)
<code>MPI_Accumulate</code>	Ready	No bubbles, no synchronization, making receive must already be finished

`MPI_Send` Standard May buffer or not, ~~can be~~ synchronous (implementation dependent)

Only one receive mode, matches all sending modes

- Orthogonal to the modes, communication can be (non-)blocking
 - Non-blocking operations return immediately, independent of the send state
 - The send buffer cannot be safely reused after non-blocking ops
 - The send buffer can be safely reused after blocking ops
 - By default MPI operations are blocking

`MPI_Sendrecv` is blocking, internally parallel, send buffers and receive buffers must be disjoint

`MPI_Sendrecv_replace` available with one buffer for send and receive

`int MPI_Isend(void *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm, MPI_Request *request)`

`MPI_Irecv(...)`

Non-blocking send and receive operations
`request` pointer to status information about operation.

`int MPI_Test(MPI_Request *request, int &flag, MPI_Status *status)`

Non-blocking check

`flag` set to 1 if operation completed; 0 if not

`int MPI_Wait(MPI_Request *request, MPI_Status *status)`

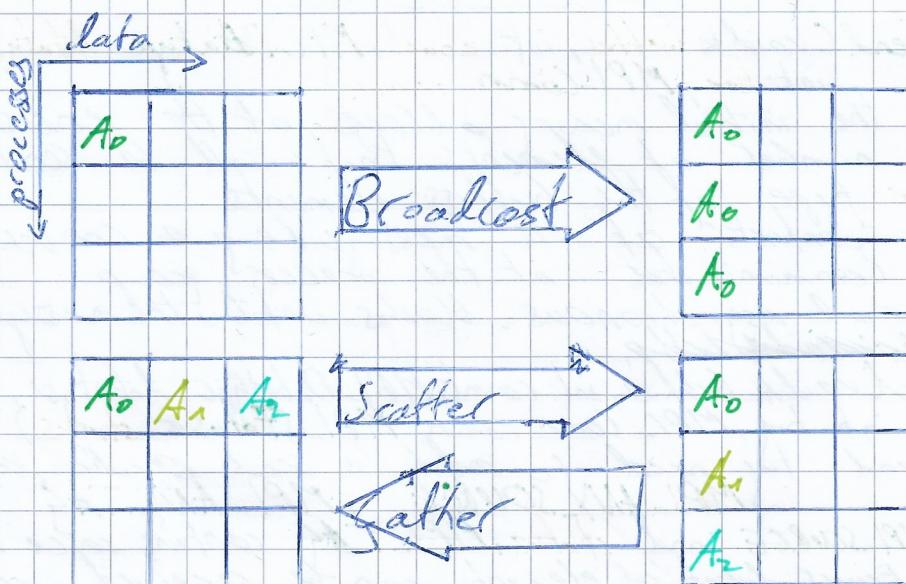
Blocking check.

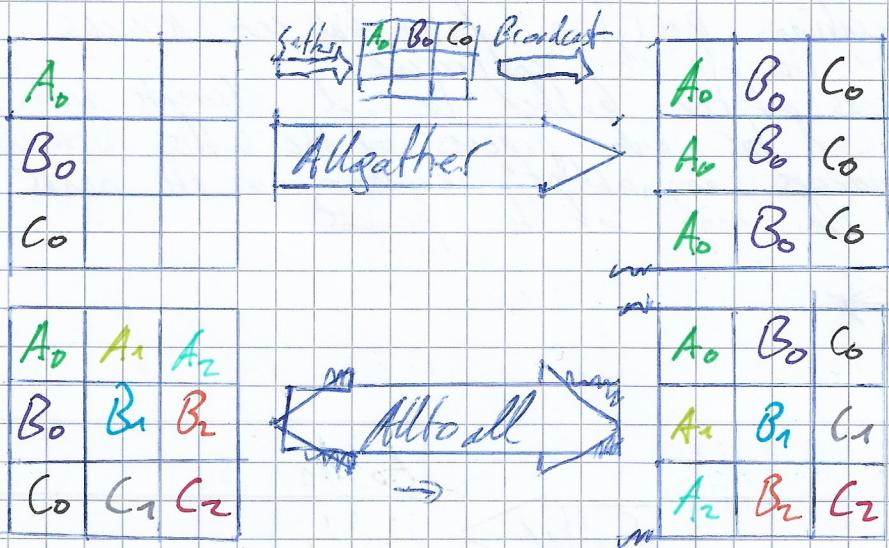
`int MPI_Bcast(void *bufs, int count, MPI_Datatype type, int root, MPI_Comm comm)`

`- root` is the origin of the message being sent

`- root uses buffer to provide data; all others use buffers to get receiving data`

`- Other parameters must be identical`





`int MPI_Scatter (void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype,
int root, MPI_Comm comm)`

All receivers get equal-sized but content-different data

`int MPI_Scatterv (void* sendbuf, int* sendcounts, int* displacements, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype, int* displacements,
MPI_Comm comm)`

Vector variant.

Allows varying counts for data sent to each process
 sendcounts integer array, entry i specifies the displacement / offset
 to sendbuf from which to take the outgoing data to each receiver
 sendcounts integer array with the numbers of elements to send
 to each process

displacements integer array, entry i specifies the displacement
 relative to sendbuf from which to take the outgoing data to process i (last allowed, but no overlap)
sendtype datatype of send buffer elements **recvtype** datatype of receive buffer elements

`int MPI_gather (void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`
 roots buffer contains collected data, sorted by rank,
 including roots own buffer contents
 recvcount number of bytes received from each process
MPI_gather vector variant

`int MPI_Mgather (void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype
recvtype, MPI_Comm comm)`

is gather + broadcast

Multicast Broadcast: for each process p; in comm: p; collects and sends the
 same data to all other processes

At the end, the buffers in each process in comm has the same data
 in the same order

MPI_Allgatherv vector variant.

`int MPI_Allgather (void* sendbuf, int sc, MPI_Datatype st, void* rcvbuf, int rcvcount, MPI_Datatype rt, MPI_Comm comm)`
 A sends ps seeds to receiver pr only its s-th element
 A receives pr stores information from sender ps at the position s in
 its buffer

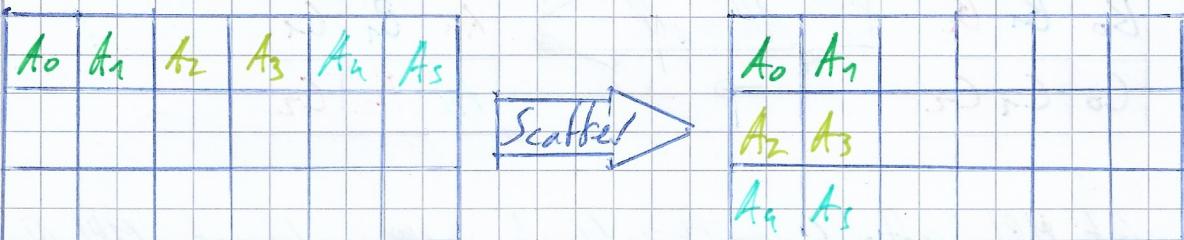
MPI_Alltoallv Separate Specification of count, displacement and datatype for each block

Prepath

- Collective operations partition data for each process
according to the size of send/receive
- Memory address of the buffer to send elements from or store them
into is calculated for each process, by the buffers initial address,
For each process an "offset" depends on the number of sub-receive
decks and the rank of the process

Offset P3

Offset P1

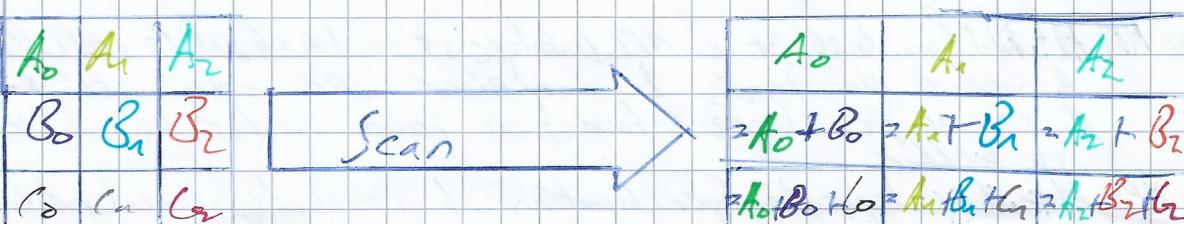
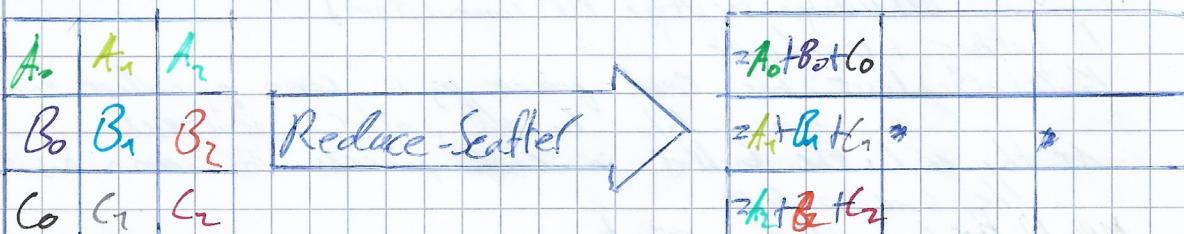
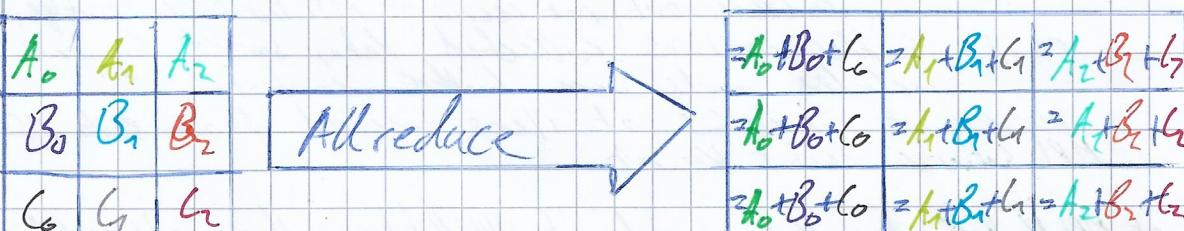
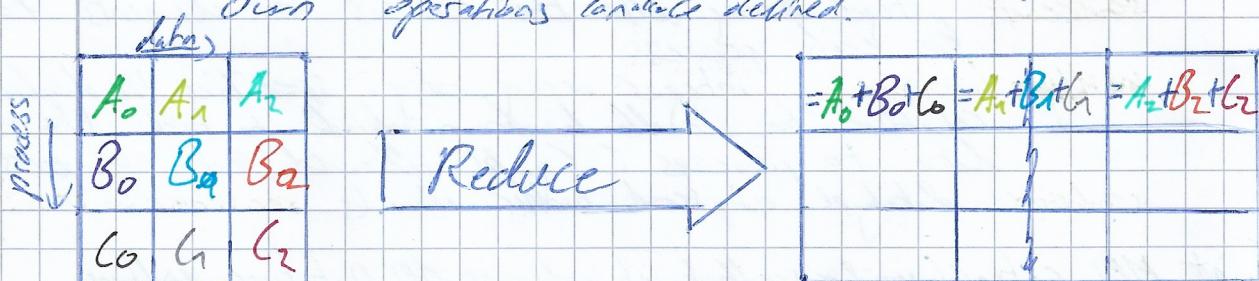


int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype type,
MPI_Op op, int root, MPI_Comm comm)

Applies an operation to the data in sendbuf and stores
the result in recvbuf of the root process
count number of columns in the output buffer

op can be

MPI_LAND logical and, MPI_BAND logical and, MPI_LOR, MPI_BOR,
MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD,
MPI_MINLOC, MPI_MAXLOC find local min-/maximum and return the
value at the "center" rank
own operations can be defined.



Operations
in descending
order

Map reduce: 1. Map 2. Shuffle 3. Reduce s. S. 4.2

Progr. 18

Java API Scatter API Reduce
Math.max(a, b), Ausdruck: Math.ceil(... oder (nennert >oder -1) / teiler, divmodIndex)
Lamdas in Java (int i, int j) → i+j oder i(j) → i+j
(new Thread(() → System.out.println("Hello"))).start();

Functional interface: interface that declares a single method serves as a target for lambda expressions

void setPriority(int priority) sets priority of thread.

Live lock: situation in which threads are not completely blocked but switch between running states making no progress

Guarded Block with Signals // synchronized(Object object) ...
waits releases monitor so another thread can enter, wants has a notify or notifyAll
throws InterruptedException
notified wakes up one thread that called wait()
notifyAll wakes up all threads that are waiting for the monitor
Only thread that owns monitor can use signals on it

volatile establishes happens-before relationship: a write to a volatile variable happens before every subsequent read

Values are not locally cached in a CPU cache
All writes to potentially different variables before writing to a volatile variable are visible to all other of that variable after reading the volatile variable

ReentrantLock (deadlock free) tryLock()

tryLock()
tryLock(long time)
tryLock(long time, TimeUnit unit)
tryLock(long time, TimeUnit unit, LockTimeoutException exception)
tryLock(long time, TimeUnit unit, LockTimeoutException exception, Supplier<Exception> exceptionSupplier)

Semaphore (int capacity, Cyclic barrier)
acquire()
release()
tryAcquire()

CyclicBarrier (int n)
await()

CountDownLatch (int n)

await() blocks calling thread
countdown() has to be called n times
for threads to continue
(cannot be reenabled afterwards)

Exchanges < V>()

good practice exchange of two objects between two threads
V exchange(V object to Exchange) has to be called by both threads and blocks until both have called it

Executor Abstract from Thread Creators

Progr 16

void execute(Runnable runnable)

ExecutorService

Subinterface of Executor

Further Lifecycle management logic

submit(Future<T> submit(Callable<T> callable))

Executors

Class, provides factory methods for creating an ExecutorService

newSingleThreadExecutor()

newFixedThreadPool(int) creates a thread pool with reused threads of fixed size

newCachedThreadPool creates a thread pool with reused threads of dynamic size

Callable<T> Allows to return results
T call()

Future<V> Represents (future) result of asynchronous computation
• get Blocks until thread finished, and Future contains the value
• get(long timeout, TimeUnit unit)
isDone Returns whether the task is finished.

CompletableFuture<T>

• supplyAsync Pass asynchronous task

• thenApply Define behaviour that is executed after the task has finished

For Down Pool on it, ForkJoinTasks can be executed

• invokeAll • parallelTask

RecursiveTask<T> • invoke() ForkSubtask • expand() Create in place Joinable

• Any Java Collection can be treated as a stream by calling the Stream API

Provide additional operators: filter, map, reduce, collect, limit, findFirst, min, max, etc.

• getAsDouble() - getAsInt(), .get()

Design By Contract

Aware triplets : {P} {C} {Q}

P: Precondition

C: Series of statements

Q: Postcondition

If P is true before the execution of either Q is true after executing C

Java Modelling Language (JML)

Pre- and postconditions defined in specialized Java comments (all lines starting with @)

requires, ensures keywords

a ==> b a implies b

a <==> b a iff b

a <!=> b !(a ==> b),

\result Result of the method call

\old(E) Value of E in the state before method execution

\forallall declaration; range-expression; body-expression

It is possible to combine conditions with either the if operator or by starting a new requires/resource line

\existsexists declaration; range-expression; body-expression

\max, \min, \sum, etc

Class invariants must hold in all user-visible states
invariant pure keine sideeffects

Prop 17

- Liskov Substitution Principle for pre- and postconditions of overriding method
 - Preconditions must ~~be~~ not be more restrictive than ^{the other} ~~overwritten~~ Precondition_{super} \Rightarrow Precondition_{sub}
 - Postconditions must be at least as restrictive as those of the overwritten methods: Postcondition_{sub} \supseteq Postcondition_{super}
 - Regarding a complete class
 - > Pre- and postconditions relations must hold for all methods as stated above
 - > The class invariants must be at least as restrictive as those of the superclass invariant_{super} \Rightarrow invariant_{super}

Compilers

Ebenen der Interpretation bzw. Übersetzung

~~Übersetzung in Maschinencode~~ (LL(1)H)

Vollständige Übersetzung: Übersetzung in Maschinencode (Big I(H))

* Reine Interpreter:liest Quelltext herunter für Ausführung und führt diese direkt aus (Unterstützung) (Bsp: Elan-Shell)

* Interpreter nach Übersetzung: Analyse für Quelle und Transformation in eine für den Interpreteren geeignete Form (Bsp: Python, Ruby,...)

* Just-in-Time-Compiler: Übersetzung während des Programmablaufs (Java, .NET)

Lexikalische Analyse:

Eingabe: Sequenz von Zeichen

Aufgabe:

Stichwort bedeutungsfolgende Zeichengruppen: Tokens

• Übergangs unbedeutende Zeichen (Pfeilzeichen, Kommentare)

• Bereichstypen definieren und Zusammenfassen in Stabtabelle

Syntaktische Analyse

Eingabe: Sequenz von Tokens

Aufgabe:

Stabtabelle, ob Eingabe zu kontextfreier Sprache gehört

• Erkennen hierarchische Struktur der Eingabe

Ausgabe: Abstrakter Syntaxbaum

Semantische Analyse

Eingabe: Abstrakter Syntaxbaum

Aufgabe: kontextsensitive Analyse

• Namensanalyse: Beziehungen zwischen Deklarationen und Verwendung

• Typanalyse: Beziehungen und prüfen Typen von Variablen, Funktionen

• Wurtsatzprüfung - alle Einschränkungen des Programmcorpus eingehalten

Ausgabe: attributierter Syntaxbaum

Originalprogramme werden späteren in syntaktisch semantischer Form abgedruckt.

Zwischencodegenerator, Optimierer

Prop 18

Aufgabe:

- Bringe Code in sprach- und zielunabhängige ZwischenSprache
Optimiere Code
- Konstantenfaltung - Bsp: Ersetze $3+8$ durch 11
 - Kopienkostschaltung
 - Code Verdichtung
 - Gemeine Teile aus dem Code entfernen
 - Inlining

zurück

Codegenerierung

Eingabe: offiziell festes Syntaxbaum oder Zwischencode
Auszug: Erzeugte Code für Zielmaschine

Anpassung an Konventionen des Gutsatzsystems

Codeauswahl - welche Befehle das Zielsystems benutzen?

Scheduling - Reihenfolge der Befehle festlegen

Registerallokation - welche Zwischenergebnisse in Prozessorregistern halten

Nachoptimierung

Ausgabe: Programm in Assemblies oder Maschinencode
durch Assembler und Binder

Syntaktische Analyse

Syntaktische Analyse (Parsing)

Verwendung von Kontextfreier Grammatik ~~um system 3.5.38%~~

Ableitungs- β zur Syntaxbaum

Grammatik ist eindeutig, wenn jedes Wort ihrer Sprache
nur einen Ableitungsbau an nimmt

AB Ableitbarer Syntaxbaum enthält

- nur spezifische Token (z.B. Id, Float) als Blätter,
- keine Schlüsselwörter als Operatoren
- Operatoren an neuen Knoten
- keine Klammern, sondern Operator-Prioritäten durch Baumstruktur
- keine Kette produktionen

LL und LR Technik: Parser liest Quelle ein und von links nach
rechts und baut dabei Links- bzw. Rechtsableitungen auf.
Fehler werden beim ersten Zeichen t , das nicht zu einem
Wort der Sprache gehören kann erkannt.

LL top down, LR bottom up, Behandelt: SIC Rekursiv Abhäng.,
entstellt von LL

Linksableitig: linke Teilbäume vor rechten

Pro Symbol der Grammatik eine Funktion, jede:

- passt & konsumiert einen Teil der Eingabe
- entscheidet anhand nächster Token, welche Variablen einzeln passen
- delegiert an Parse-Funktionen ansetzt Symbole

Rekursiver Rüstling muss Produktion mit k-Look-ahead unterscheiden

h -Anfang $h : x$

Für $x \in \Sigma^*$ ist der h -Anfang $h : x$ das Präfix der Länge h von $x \# \dots$, wobei $\# \notin D$ das Ende des Terminalbogens kennzeichnet.

Für $x \in (\Sigma \cup V)^+$ ist First_h definiert als

First_h-Menge

$$\text{First}_h(X) = \{\beta \in \Sigma^* \mid \exists t \in \Sigma^* : X \Rightarrow^* t \wedge \beta = h : t\}$$

First_h(X) sind die h -Anfänge der Strings, die aus X generiert werden können.

First und Follow_h

Für $X \in (\Sigma \cup V)^+$ definiert

$$\text{First}_h(X) = \{\beta \mid \exists t \in \Sigma^* \text{ mit } X \Rightarrow^* t \wedge \beta = h : t\}$$

$$\text{Follow}_h(X) = \{\beta \mid \exists \alpha, w \in (V \cup \Sigma)^* \text{ mit } S \Rightarrow^* \alpha X w \wedge \beta \in \text{First}_h(w)\}$$

Follow_h

Follow_h(X): h -Anfänge der Strings, die hinter X generiert werden können.

Für Produktion $A \rightarrow \alpha$ ist die Indizmenge der an dieser Stelle beim Parsen potentiell sichtbaren (h -)Lookheads, genau First_h(α) Follow_h(A))

Eine Kontextfreie Grammatik ist genau dann eine ~~SLL(h)~~ SLL(h)-Grammatik wenn für alle Paare von Produktionen $A \rightarrow \alpha \mid \beta$, $\alpha \neq \beta$ gilt $\text{First}_h(\alpha) \cap \text{First}_h(\beta) = \emptyset$

Spezialfall $h=1$, $\alpha \Rightarrow^* E$, $\beta \Rightarrow^* E$ $SLL \Leftrightarrow \text{First}(\alpha) \cap \text{First}(\beta) = \emptyset$

Spezialfall $h=1$, $\alpha \Rightarrow^* E$, $\beta \Rightarrow^* E$ $SLL \Leftrightarrow \text{First}(\alpha) \cap \text{First}(\beta) = \emptyset$

gg SLL(h) \Leftrightarrow g mit h -Lookahead in rekursiver Abstieg passbar

Linksseitiger Greifversuch: Verzage $X \rightarrow \gamma \alpha \beta$, verzage Entscheidung zw. α und β bis nachdem γ konsumiert wurde

$$\begin{aligned} X &\rightarrow \gamma X \\ X &\rightarrow \alpha \beta \end{aligned}$$

Linksseitige kontextfreie Grammatiken sind für kein ~~SLL(h)~~ SLL(h)

Fazitregeln Grammatik Eigenschaften

- Ein Nichtterminal pro Prioritätsstufe
- Häufig Schichtung: Produktionen von niedrig zu hoch Priorität
- dasselbe Nichtterminal maximal auf der Rechten Seite
- Assoziativität von Operatoren: Wie ist $a + b + c$ zu fassen?
 $(a + b) + c \Rightarrow$ Linksgreifversuch $E \rightarrow E + T$
 $a + (b + c) \Rightarrow$ Rechtsgreifversuch $E \rightarrow T + E$

SLL(h)-Grammatik für möglichst kleines h (effizienter)

Grammatik eindeutig & Assoziativität durch grammatisch vorgegebene

Implementierung von ASTs

für jedes syntaktischen Kategorien eine Klasse

für Blätter und Knoten in Kategorien in Konsistenz verarbeiten

Rekursives Abstieg: Linkssozialitätstot s. S. 414

Pragmatik

Semantische Analyse

Worauf kann SIMPLE und Java:

- Namensraum für Variablen & Parameter gebunden von Namensraum der Prozeduren
- Bei Prozeduren Verwendung von Definition möglich
- Zusätzlich: verschachtelte Namensbereiche durch Blöcke

Implementierung der Namensanalyse:

Vorgehen: Durchlaufe AST und summe Informationen in Symboltabelle

Symboltabelle

- Enthält Informationen zur (momentanen) Definition von
- Unterstützt verschachtelung von Namensbereichen Bezeichnern
- Optional Unterstützung getrennter Namensräume

- Ansatz:
- Einträge in Symboltabelle verweisen auf momentane Definition
 - Stack von Namensbereichen: Jeder Bereich enthält Liste von Definitionen
 - Bei Definition eines Bezeichners muß vorher Definition in Liste
 - Beim Verlassen eines Namensbereichs stellt vorherige Definitionen anhand des Liste nichts her

AST - Transposition Nutze visitors pattern

Java Bytecode

Virtuelle Maschine

Grund

Method Area: Code der Methoden nutzbar

Runtine Constant Pool

Threads: Da Thread:

Program Counter

JVM Stack

Native Method Stack: Für Laufzeitssystem

Stack basierter Bytecode: Gründlos und Rückgratlos wegen der dynamischen Größe

Instruktionen

explizit für C/C++ add(int), add(float)

Instruktionstypen

- Lesen/Schreiben von lokalen Variablen (load, store, ...)
- Lesen/Schreiben von Feldern (getfield, putfield, ...)
- Sprungbefehle (ifeq, ifne, tableswitch, ...)
- Methodenauswahl (invokenew, invokestatic, ...)
- Objektzeugen (new, newarray, ...)
- Arithmetische Berechnungen (imul, iadd, ...)

Methodenaufrufe

1. This-Parameter pn auf den Operandenstack (falls nicht statisch)
2. Parameter $param$ auf Operandenstack
3. mehrwertig / instanzstatisch ausführen - Automatisch prüft
 - a. Wenn Activation Record erzeugt
 - b. Rückgradausgabe (Program Counter) auf und dynamischen Framepointer (Framepointer) in Activation Record sichern
 - c. Neuen Framepointer setzen
 - d. Parameter (pn, \dots, pn, \dots, pn) von Operandenstack in neuen Activation Record kopieren
 - e. Zur Methodenaufrufe springen
4. Block wird ausgelöst
5. Rückgabewert auf den Operandenstack
6. Rückgabewert ausführen (return, break, ...) automatisch gesetzt
7. Alten Framepointer setzen und zur Rückgradausgabe
 - a. springen
8. Rückgabewert aus Operandenstack des neuen Activation Block in Werte Operandenstack hinzun.

Descriptors

- Namen von Typenfeldern und Methoden muss festgelegtes Schema entsprechen
- Objekttypen $java.lang.Object \rightarrow$ Ljava/lang/Object;
 - primitive Typen int, float, void \rightarrow I, F, V
 - Methoden void foo(int, Object) \rightarrow foo(ILjava/lang/Object;)V
 - Descriptors: (Parametertypen) Rückgabefall, Identifikator des Name & Descriptor
 - Felder: boolean b \rightarrow Z Identifikator nur des "Name"
 - Konstruktoren: Name ist <init>, static Initializer <CLinit>

Objekt erzeugen & initialisieren

1. Objekt anlegen \rightarrow Speicher reservieren

2. Objekt initialisieren \rightarrow Konstruktor aufrufen

Umgekehrte polare Methode (UPN) zuerst ergründen, dann ausführbare Operation

7*4 in UPN 74*
 2*(2+3) in UPN 223+*
 5+3*5 in UPN 59+35*+

Befehlserzeugung für arith. Ausdrücke
Abstrakter Syntaxbaum

Postfixordnung für Tiefensuche erzeugt UPN

Postfixordnung Befehlserzeugung beim Wählen eines Knotens (durch den UPN-Knotenfolge integriert Bytecode-Befehle selektive Anwendung sind)

Kontrollfluss

- Jeder AST-Knoten ruft rekursiv codeGen() Methode seiner Kindheit
- Jede Konditionsstruktur generiert ihren Code so, dass sie nur einen Fall durchverlassen wird
- \Rightarrow Konditionsfall ausgelöste codeGen() methoden erzeugen nodeinits angeleiteten Code
- Code erzeugt bei Bedingungen (z.B. $<=$) Schleife zw. Sprungzelle übergibt, zu deren dem zu erzeugte Ende je nach Laufzeitwert der Bedingung springt.

W. Kondition (if t) Erzeuge Code 1 das ausgeführt wird, falls rechte Seite ausgewählt wurde und

Prob. 22
Negation: Nein Zusätzlicher Bytecode senden refusnug der Sprungzelle

Java Bytecode

? load < x > Lädt Variable x (-> value)
? store < x > Speichert Variable x (Value ->) .
ldc < x > Ladt Wert x (-> value)

? mul Multipliziert oberste 2 Stackelemente (val1, val2) -> result
? add

? sub

? inc < x > < y > erhöht Variable x um Wert y

ifeq label Branch zu Label, wenn Stackwert 0 (val ->)
ifnull null (val ->)
ifle ≤ 0 (val ->)

getfield index Belohmen Field value by index im Const pool (0 get -> value)
putfield index Setze

Stack value

(objectref, value ->)

New index Neues Objekt von index im Const pool (-> 0 get)
newarray atype Neues array von Typ atype (count -> arrayref)

goto label

? return return ? from a method (? -> [empty])

dup Dupliziere value auf Stack (val -> val, val)

indirect

invokenv rufe virtuelle Methode auf, ergänzt at stack, index ab Const pool

static statische

special instance

if_icmpne v1 = -> v2

if_icmpge ≥

gt >

le ≤

lt <

ne ≠