

# Propa Zusammenfassung

Haskell: ~~noch~~

Linear Rekursiv: In jeder Definition zweig kommt nur ein rekursiver Aufruf  
Endlich: Linear Rekursiv, in jeder Zweig ist der rekursive Aufruf  
nicht in andere Aufrufe eingebettet.

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

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

Funktionskomposition  $\circ g$ :  $\text{comp } f g = ((x \rightarrow f(g x)) \text{ hält } f \circ g$   
 $n$ -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 k =  $x * x$

in m \* (square c)

energy m = m \* (square c)

where c = 299

square x =  $x * x$

Einrückung hat semantische Bedeutung.

Bei Schleifenzug: Inneres let bindet stärker

Folds:

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

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

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 elementbedingung Liste, beginn b (= takeWhile b, rest)  
Verzerrung Intervalle  $[a .. b] \Rightarrow [a, a+1, a+2, \dots, b]$   
 $[a .. ]$  unendl. Intervall, start nicht a

List Comprehensions  $[c | q_1, \dots, q_m]$ , q\_i Tests oder Generatoren  
der Form  $P \in \text{list}$   
mit Klammer für Listenausdruck list

Beispiel etwas  $n = \{x \mid x \in [0, n], \underline{x \bmod 2 = 0}\}$

Kap 2

(Inendliche Listen: odds =  $\text{iterate } (\lambda x : \text{map } (+2) \text{ odds})$ )

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

odds = iterate (+2)

iterate  $f x \cdot !! 23$  führt "Schleife" 23 Mal aus repeat == iterate ( $\lambda x \cdot x$ )

type readName = [Integer]  
= String  
= Char

• Algebraische Datentypen: dekorativ und klar konstruieren

data Shape = Circle Double

Rectangle :: Double -> Double -> Shape

1 Rectangle Double Double

Circle :: 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  
1 Sparse [(Integer, Integer, t)] t - Einträge  $(i, j, v)$  und  
Oberflächen.

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

Polymorphe Typen Funktion mit Typengeschränkung

qsort :: Ord t => [t] -> [t] instanzieren t von Ord implementieren  
qsort [] = []  
qsort (p:ps) = ...

Eigenschaften-Deklarationen

instance Eq Bool where

True == True = True

True == False = False

Fehlende Implementierungen: Default implementieren

data Shape = Circle Double

1 Rectangle Double Double

defining Eq keine eigene (==) Funktion mehr notwendig,  
Automatische Instanziierung durch List Show, Ord, Enum

Moraden

# untypisierte $\lambda$ -Kalkül

	Notation	Bespreche
Variablen	$x$	$x$
Abschaltung	$\lambda x. t$	$t[x \rightarrow \dots]$
Funktionsanwendung	$t_1 t_2$	$(\lambda x. t_1) t_2$ $(t_1 x) t_2$ $(t_1 x) t_2$

Funktionsanwendung linksassoziativ, bindet stärker als Abstraktion

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

$\eta$ -Äquivalenz: Terme  $\lambda x. t$  und  $t$  heißen  $\eta$ -äquivalent  $(\lambda x. t) \eta \equiv t$  falls  $x$  nicht freie Variable von  $t$  ist

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

$\beta$ -Reduktion:  $\beta$ -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.

Normalform: Ein Term, der nicht weiter reduziert werden kann

Volle  $\beta$ -Reduktion: Jeder Redex kann gezielt reduziert werden

Normalbegriffstafel: immer der linke äußerste Redex wird reduziert

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

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

$$c_0 = \lambda s. \lambda z. z$$

$$c_1 = \lambda s. \lambda z. s z$$

$$c_2 = \lambda s. \lambda z. s(s z)$$

$$\vdots$$

$$c_n = \lambda s. \lambda z. s^n z$$

Substitution: sub

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

Multiplication: times =  $\lambda m. \lambda n. \lambda s. s(m s)$

$$\vdash \lambda m. \lambda n. \lambda s. s(m s) \#$$

Potenzieren: exp =  $\lambda m. \lambda n. n^m$

$$\vdash \lambda m. \lambda n. \lambda s. s(m s) \#$$

true =  $\lambda t. \lambda f. t$  false =  $\lambda t. \lambda f. f$

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

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-Rosser

Wenn  $t \xrightarrow{*} t_1$  und  $t \xrightarrow{*} t_2$   
dann gibt es  $t'$  mit  $t_1 \xrightarrow{*} t'$  und  $t_2 \xrightarrow{*} t'$

Call-by-name: Reduzierung ~~aller~~ linker äußerster Redex  
Abes nicht falls von einer  $\lambda$  umgeben  
 $\rightarrow$  Reduzierbare Argumente ~~sind~~ nur benötigt.

Call-by-value Reduzierung ~~aller~~ Redex der nicht von einem  $\lambda$  umgeben  
und dessen Argument ein Wert ist  
 $\rightarrow$  Argumente von Funktionsaufruf auswerten.

## Regelsysteme:

Freigeschafftes Schlossstück:

$q_1 \ q_2 \ \dots \ q_n$

$\varphi$

Jede Regel stellt  
Implikation dar,  
 $q_i$  Voraussetzung,  
 $q$  Konklusion

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

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

$$\forall E \frac{\forall x. P(x)}{P(y)}$$

Hilfestellung genügt  $\varphi$

Regeln verbindet

Implikation der Regeln mit Hilfestellung  $\varphi$

Implikation  $\rightarrow I \varphi \rightarrow \varphi$

$$\mu \varphi \frac{\varphi \rightarrow \varphi}{\varphi}$$

## Alternative Notation

$$\frac{\Gamma, t \vdash q_1 \ \dots \ \Gamma, t \vdash q_n}{F \vdash p}$$

$\Gamma$ : Liste von Aussagen, gen. Kontext  
 $\Gamma \vdash q$ :  $q$  unter Annahmen  $\Gamma$  herleitbar

$\rightarrow$

→ Regelsysteme alternative Notation:

Introduktionsregeln

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

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

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

...

Eliminationsregeln

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

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

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

## Typsysteme

Funktionsarten einstassoziativ

Typsystem  $\Gamma \vdash t : \tau$  im Typkontext  $\Gamma$  hat Form  $t$  der Typ  $\tau$   
 für alle freien Variablen  $x$  ihren Typ  $\Gamma(x)$  zu.

$$\text{CONST } \frac{}{\Gamma \vdash c : \tau_c}$$

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

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

$$\text{APP } \frac{\Gamma \vdash t_1 : \tau_1 \rightarrow \tau \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1 t_2 : \tau}$$

Typisierung von  $\lambda$ -Termen: Paar  $(\Gamma, t)$ , sodass  $\Gamma \vdash t : \tau$  lehrbar  
 $\alpha$ -Reduktion: Substitution von  $\alpha$  ( $\lambda x. t_1 t_2 \Rightarrow t_1[x \mapsto t_2]$ )

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

-Typschaltung: Wenn  $\Gamma \vdash t : \tau_0$  und  $t \Rightarrow t'$  dann  $\Gamma \vdash t' : \tau$

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

Instanzierung eines Typeschemas: Für nicht-Schematypen  $\tau_2$  ist der Typ  $\forall x. \alpha \rightarrow \tau_2$  eine Instanzierung vom Typeschemma  $\forall x. \alpha$   
 Schreibweise  $(\forall x. \alpha) \Sigma \tau_2[\alpha \rightarrow \tau_2]$

Beispiele: Beispiel  $\forall x. \alpha \rightarrow \alpha \vdash \text{int} \rightarrow \text{int}$  Aber:  $\alpha \rightarrow \alpha \not\vdash \text{int} \rightarrow \text{int}$

$\forall x. \alpha \rightarrow \alpha \vdash \text{list} \rightarrow \text{list} \rightarrow (\text{int} \rightarrow \text{int})$ :

$\text{int} \not\vdash \text{int}$

$\alpha \not\vdash \text{bool}$

$\forall x. \alpha \rightarrow \alpha \not\vdash \text{bool}$

Angepasste Regeln

$$\text{# VAR } \frac{\Gamma(x) = \tau' \quad \tau' \neq \alpha}{\Gamma \vdash x : \tau}$$

$\Gamma, x : \tau_1 \vdash t : \tau_2$   $\tau_1$  kein Typeschemma

ABS:  $\Gamma \vdash \lambda x. t : \tau_1 \rightarrow \tau_2$

Constant für Regeln bleiben gleich

$\lambda$ -gebundene Bezeichner werden polymorph

Propa 6

Typabstraktion: Das Typschema  $\text{ta}(\bar{x}, \Gamma) = \text{Dom}_1 \times \dots \times \text{Dom}_n$  heißt Typabstraktion von  $\Gamma$  relativ zu  $\Gamma$ , wobei  $\bar{x} \in \text{FV}(\bar{c}) \setminus \text{FV}(\Gamma)$   
 $\Rightarrow$  Verhindern Abstraktion von globalen Typvariablen im Schema

$$[\Gamma \vdash t_1 : \bar{c}_1 \quad \Gamma, x : \text{ta}(\bar{c}_1, \Gamma) \vdash t_2 : \bar{c}_2]$$

LET:

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

Haller Praktikum

Terme: Atome:  $\text{hang}$ ,  $\text{inge}$  beginnen mit Kleinbuchstaben  
 Zahlen:  $3, 4, 5$   
 Variablen:  $X, \underline{X}$  beginnen mit Großbuchstaben oder Unterstrich.  
 Term Listen:  $3, 4, 5, X, \underline{X}$   
 Zusammengesetzte Terme:  $\text{lebt}(\text{fratz}, \text{high})$ ,  $\text{lebt}(\text{fratz}, X)$

Atom an Stelle eines zusammengesetzten Terms: Funktionsaufrufe funktionale Funktionen folieren in Infixnotation ( $3 < 4$ )

Atome und variablenechte Terme stehen für sich selbst

Variablen sind Platzhalter für unbekannte, Terme andern Regelwerk  
 Fächer implizit universell quantifiziert.

Ablöse durch Projektion? eingeschränkt und mit Punkt beendet.  
 Erhältbar geht nicht erfüllbar.

Ablöse Ablöse mit Variable: versucht Ablöseforsa mit Daten durch zu unterscheiden. Konjunktion von Teilzielen getrennt durch Komma

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

Teilziel erfüllt von links nach rechts  $\Rightarrow$  passende Instantienungen  
 Nach Erfüllung eines Teilziels: nächstes Teilziel ist instanziiert.

Regeln:  $\text{term} :- \text{termList}.$       :- Als "Wear" zu legen.  
 „Wenn term ist, dann term“

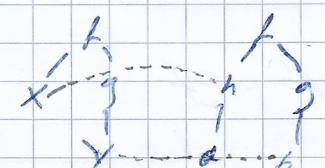
Vorabellen in Regelkopf implizit universell quantifiziert. existentiell  
 Variablen die ausschließlich in Regelkörpfer vorommen implizit universell quantifiziert

Gruppe von Fächer/Regeln mit gleicher Funktior und gleicher Argumentzahl in Regelkopf heißt Prozedur oder Fach

Beispiel:

$\text{grandparent}(X, Y) :- \text{parent}(X, Z), \text{parent}(Z, Y).$        $\text{parent}(X, Y) \wedge \text{parent}(Z, Y) \rightarrow \text{grandparent}(X, Y)$   
 $\text{parent}(X, Y) :- \text{mother}(X, Y).$        $\text{mother}(X, Y) \rightarrow \text{parent}(X, Y)$   
 $\text{parent}(X, Y) :- \text{father}(X, Y).$

$\text{mother}(\text{inge}, \text{emil}).$



$$f(X, g(Y)) = f(h(a), g(b)) \text{ mit } X \mapsto h(a) \quad Y \mapsto b$$

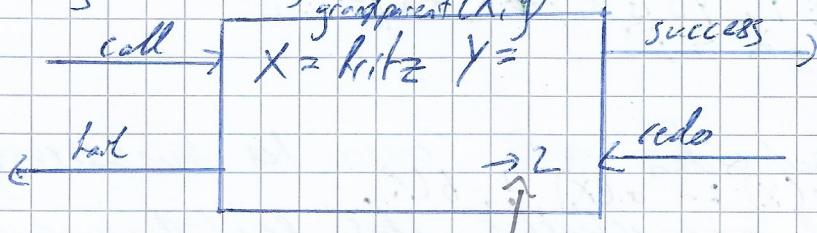
Umstitution: Term (intervallweise dargestellt):

Prop 7

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

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

Bachtraching: Visualisierung des Suchbaums. Jedes Teilziel des Baums



Choice Point: nächste zu probierende Regel bei Rechteilungsversuch.

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

Informeller Algorithmus auf Slide 24.1

Listen

$[X|Y]$ : erster 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) vor vorne aufgebaut werden

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

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

delete( $[L_1, X, L_2]$ )  $\rightarrow$  Deletes  $X$  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  $L$  die Liste  $T$  ergibt, dann ergibt die Konkatenation von  $[X|R]$  und  $L$  die Liste  $[X|T]$ .

revers rev( $X, Y$ ) :- 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 Pfeil müssen instanziert sein.

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

Arithmetische Ausdrücke unifizieren nicht mit Konstanten

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)

Programm 8

Generate and Test

erzeugen von Lösungskandidaten und welche danach getestet werden.

nat(0).

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

) Unendlich oft erstellbar.

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

$\begin{cases} Y \geq X, Y \leq Y^2 \\ (Y+1)^2 \geq X \end{cases}$

Cut: ! Abscheiden von Teilen des Ausführungsbaums

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

Als Testziel immer erstellbar, aber Regelmäßigkeitsregeln lässt alle Teilzeile links vom Cut sofort fehlschlagen.

Blauer Cut beeinflusst, weder Programmablauf noch -verhalten.  
Grüner Cut beeinflusst Programmablauf, 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 als Instanz geprüft?  
atomic(X) ist X mit einem Atom oder einer Zahl instanziert?

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

D = [ (1, [ich, bin]), (2, [ich, denke]) ] .

Lookup(N,D,A) Instantiiert D partiell nach A, funktionsfest  
Definition Folie 281.

$x = y$  unifizisch: x mit y schlägt fehl, falls unmöglich  
 $x = z \wedge y = z$  erholgisch, falls X und Y bereits unifiziert wurden.  
 $x = y$  gleichsäugiger Vergleich, X und Y müssen instanziert sein  
 $x_2 .. L$  Konstruktion/Zerlegung von x aus/in Liste L

Freeze: Tests sollen automatisch so leicht wie möglich ausgeführt werden

freezeAll(CC(X1, X2, ..., XT))

Wertet Teilziel T test aus, wenn alle X1, X2, ... instanziert sind.

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

L ist Liste von Variablen L

# Unifikation und Resolution

Prop 9

Gegeben: Menge  $C$  von Gleichungen über Termen, etwa

Typusme  $\Gamma = \frac{\text{BasisTyp}}{\text{int. bdd}}$  | Var  
 $a_1, a_2, b_1, \dots$  |  $\frac{\text{Gleichung}}{\text{Gleichung}}$   
z-stelliger Typenstruktur

Prologtume  $\Theta = \frac{\text{Atom}}{\text{Folge}, \dots}$  | Var  
 $x_1, y_1, \dots$  | Atom( $O_1, \dots, O_n$ )  
 $n$ -stellige Funktion

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

Allgemeinstes Unifikator (most general unifier or mu)  
 $\sigma$  mu ist, falls  $\sigma$  Unifikator  $\forall \exists$  Substitution  $\delta$ :  $\delta = \sigma\theta$

Unifikationsalgorithmus auf slide 299

Robinson (d. Muñoz, Paterson) - Weg von Unifikator  $\Rightarrow$  slide 302

Resolutionsregel

$\Gamma_1, \sigma_1, \dots, \Gamma_n, \sigma_n$  Fazme;  $\alpha := \alpha_1, \dots, \alpha_n$  eine Regel;  $\sigma$  mu von  $\alpha$   
 $\sigma(\alpha_1), \dots, \sigma(\alpha_n), \sigma(\theta_1), \dots, \sigma(\theta_n)$

Spezialfall u Fakt "h = c"

Start: Liste der  $\alpha$  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

$(\Gamma_1, \Gamma_2, \dots, \Gamma_n, \gamma)$  Fazme plus Substitution;  $\alpha := \alpha_1, \dots, \alpha_n$  eine Regel;  
 $\sigma$  mu von  $\alpha$  und  $\gamma(C)$   
 $(\alpha_1, \dots, \alpha_n, \theta_1, \dots, \theta_n, \sigma\gamma)$

- $\gamma$  ist die Substitution, die bei den vorherigen Resolutionsgeschritten berechnet wurde
- Wird die Liste der Teile leer ( $\emptyset$ ), wird  $\gamma$  ausgegeben.

Resolutionprinzip

Sei  $P$  ein logisches Programm (i.e. eine Liste von Fakten und Regeln). Sei  $\Gamma_1, \dots, \Gamma_n$  eine Liste von Zielen (Fazmen)

1. Die Schreibweise  $P \Gamma_1, \dots, \Gamma_n$  bedeutet: Die Ziele  $\Gamma_1, \dots, \Gamma_n$  legen sich mittels Resolution abarbeiten (mittels herkömmlicher Regeln von  $P$  in die leere Zielliste verwandeln)

2. Die Schreibweise  $P \Gamma$  bedeutet: Die Zielliste  $\Gamma_1, \dots, \Gamma_n$  ist die logische Konsequenz aus den Fakten und Regeln des Programms  $P$ .

Korrektheit und Vollständigkeit: slide 309 u. 310.

## Type inference: λ-Calculus

Vorgegeben: Form  $(\Gamma, t)$  Problem: Finde Lösung  $(\sigma, \tilde{t})$

Vorgeben:

① Erstelle Herleitungsbaum anhand symbolischer Struktur und Typisierungsrègeln, Verwende zunächst überall freie Typvariable  $\alpha_i$

② Extrahiere Gleichungssystem  $C$  für die  $\alpha_i$  gemäß Regeln

③ Bestimme allgemeinsten d.h. Wahr  $\sigma$ , das Gleichungssystem mit Lösung:  $(\sigma, \tilde{t}[\sigma])$  wobei  $\sigma_i$  die für  $t$  passende Typvariable

Wirkungsalgorithmus: init $(C) =$

```

if C == Ø then C
else let {  $\tilde{\sigma}_1, \tilde{\sigma}_2$  } U  $C' = C$  in
    if  $\tilde{\sigma}_1 = \tilde{\sigma}_2$  then unity( $C'$ )
    else if  $\tilde{\sigma}_1 = \alpha$  and  $\alpha \notin FV(\tilde{\sigma}_2)$  then unity( $C[\alpha \mapsto \tilde{\sigma}_2] \cup C'$ )
    else if  $\tilde{\sigma}_2 = \alpha$  and  $\alpha \notin FV(\tilde{\sigma}_1)$  then unity( $C[\alpha \mapsto \tilde{\sigma}_1] \cup C'$ )
    else if  $\tilde{\sigma}_1 = (\tilde{\sigma}_1' \rightarrow \tilde{\sigma}_1'')$  and  $\tilde{\sigma}_2 = (\tilde{\sigma}_2' \rightarrow \tilde{\sigma}_2'')$ 
        then unity( $C' \cup \tilde{\sigma}_1' = \tilde{\sigma}_2', \tilde{\sigma}_1'' = \tilde{\sigma}_2''$ )
    else fail
  
```

Typinference in Prolog S. 331 / Let-Polytypismus in Prolog S. 340

Type inference für Let

- clarifizierbar  $\sigma_{let}$  für Let aus linkem Teilbaum bestimmen
- Weiter im rechten Baum mit  $\Gamma' = \sigma_{let}(\Gamma)$
- Vereinfachung  $C_{let} = \{\alpha_i = \sigma_{let}(\alpha_i)\}$  Let definiert für  $\alpha_i$
- erstellt Constraints  $\Gamma'$  restliche Typinferenz  $\sigma_{let}(\Gamma')$

Allgemeine Vorgehensweise:

1. Sei  $C_0$  die leere Constraintmenge, i.h.  $\{\alpha_i = \alpha_j\}$

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

$$\begin{array}{c} \dots \\ \Gamma \vdash t_1 : \alpha_2 \\ \dots \\ \Gamma \vdash t_2 : \alpha_3 \\ \hline \Gamma \vdash \text{let } \alpha_2 \text{ by } t_2 : \alpha_1 \end{array}$$

3. Berechne den mögl.  $\sigma_{let}$  von  $C_{let}$

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

5. Berechne  $\Gamma'$  im rechten Teilbaum, sammle Constraints in  $C_1$

6. Ergebnisconstraints sind  $C_0 \cup C_{let} \cup C_1$  mit

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

## Parallel Programming

Flynn's Taxonomie

1. SIMD Single Instruction x Single Data

von Neumann architecture, e.g. single, uniform stream operations on a single data item

2. SIMD Single Instruction x Multiple Data

e.g. instruction is applied to homogeneous parallel eg. vector process

3. MIMD Multi Instruction x Multiple Data

different processors operate on different data e.g. multiprocessor

4. MISD Multiple Instruction x Single Data

multiple instructions can work on the same data e.g. pipeline

# Amdahl's Law

Pragm 11

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

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

Maximal speedup with parallel processing, parallelizable percentage of the program

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

## Message Passing Interface (MPI)

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

Communicators

• Befehl • Befehl wird  
schlagwörtig

- 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  $R_j$  ( $0 \leq R_j < N$ )
- inf\_size, rank, MPI\_Comm MPI\_Comm\_rank (MPI\_COMM\_WORLD, &my\_rank)** return the number of the rank of the process 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 can be released  
**int MPI\_Probe (MPI\_Status &status, MPI\_Comm comm)**

**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, wildcard possible  
**MPI\_ANY\_SOURCE, MPI\_ANY\_TAG**

**status, MPI\_SOURCE and status, MPI\_TAG** containing source and tag  
- **count**: Fewer datatype elements can be received. More would be ignored

**MPI\_PROBE** value can be used, to receive message of unknown 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

<code>MPI_Ssend</code>	No bubbles, send operations
<code>Synchronous</code>	No bubbles, synchronization (both sides wait for each other)
<code>MPI_Bsend</code>	Explicit buffering, no synchronization (no wait for each other)
<code>Buffed</code>	No bubbles, no synchronization, matching receive must already be initiated
<code>MPI_Asend</code>	No bubbles, no synchronization, matching receive must already be initiated
<code>Ready</code>	No bubbles, no synchronization, matching receive must already be initiated
<code>MPI_Ssend Standard</code>	May buffer or not, <sup>can be</sup> 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
  - All 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/receive

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

`MPI_Recv(...)`

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)`

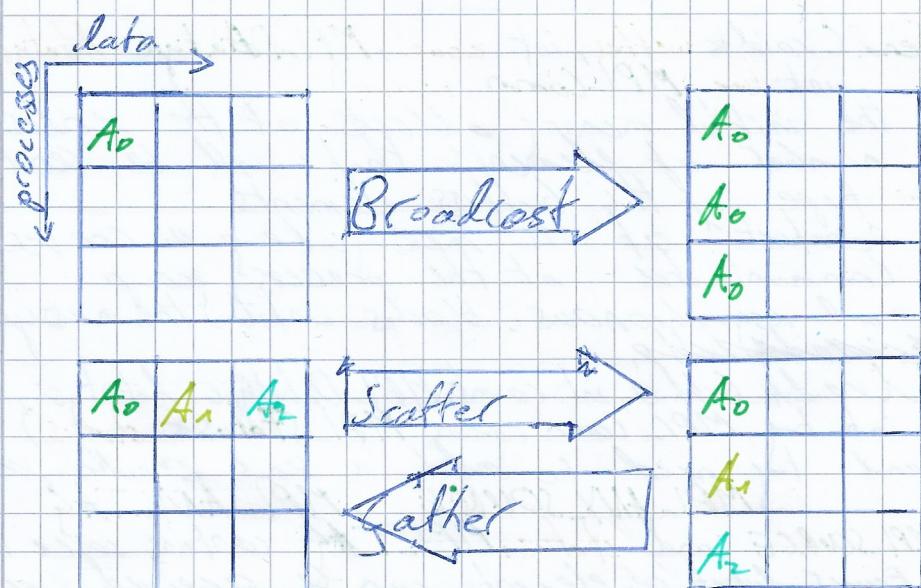
Blocking check.

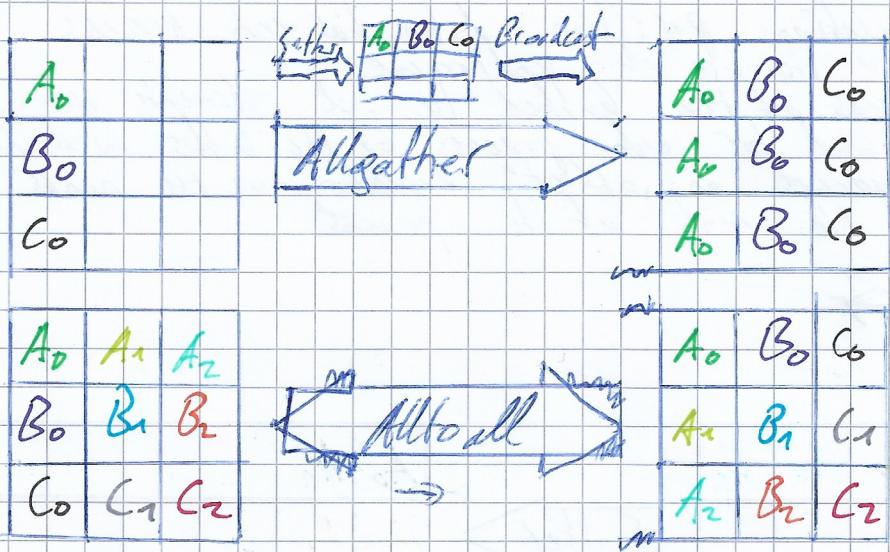
`int MPI_Bcast(void *buf, int count, MPI_Datatype type, int root, MPI_Comm comm)`

`root` is the origin of the message being sent

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

Others parameters must be identical





Property D  
w

`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

~~displacements~~ integer array, entry i specifies the displacement / offset  
to sendbuf from which to take the outgoing data to first  
sendcount integer array with the number 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\_gatherv vector variant

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

is gather + broadcast

Multi-Broadcast: for each process  $p_j$  in comm:  $p_j$  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_Alltoall (void* sendbuf, int sc, MPI_Datatype st, void* rc, int rc, MPI_Datatype rt, MPI_Comm comm)`  
A sender  $p_s$  sends to receiver  $p_r$  only its  $s$ -th element  
A receiver  $p_r$  stores information from sender  $p_s$  at the position  $s$  in  
its buffer

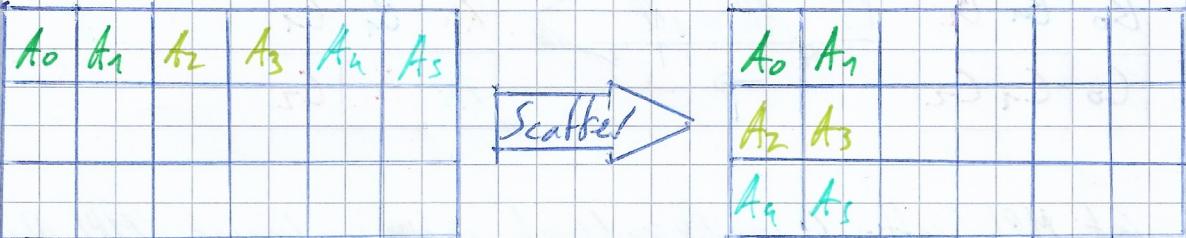
MPI\_Alltoallv Separate specification of count, displacement and datatype for each block

- Collective operations partition data for each process  
according to the use of global memory
- 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" depending on the ranks of each rank, demands and the rank of the process

Repath

Offset Pg

Offset Pm



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

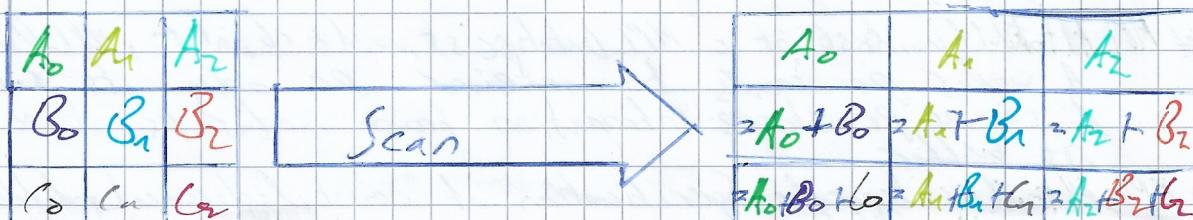
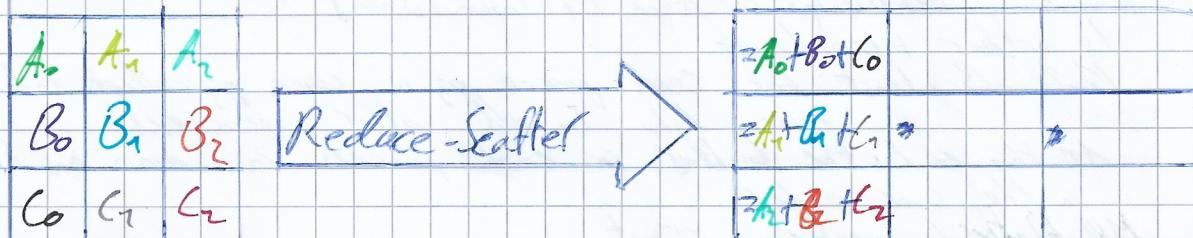
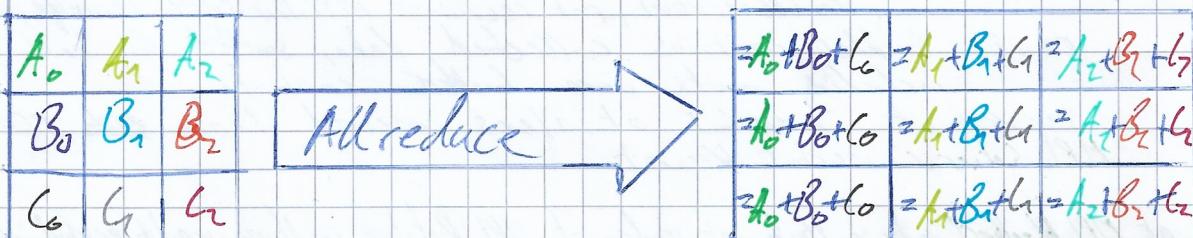
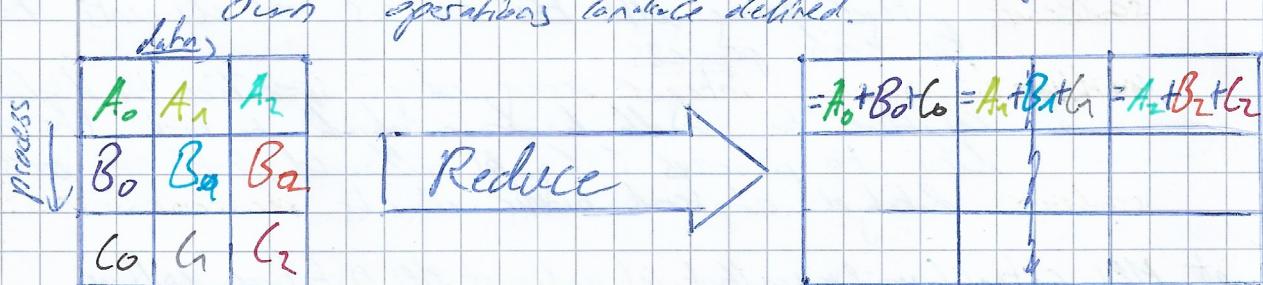
Applies an operation to the data in sendbuf and stores the result in recvbuf of the root process

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.



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

Progr 18

Java API Scatter API Reduce  
Math.max(a,b), Aufruender: Math.ceil(...).oder (nennert >oder -1) / taller, Linieneinde  
Lamda 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) ...  
**wait()** releases monitor so another thread can enter,  
wants has a notify or notifyAll  
throws InterruptedException

**notify()** 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 belong  
writing to a volatile variable are visible to all other  
of that variable after reading the volatile variable

ReentrantLock (deadlock free) tryLock()

try

acquire()

unlock

lock

try

finally

unlock

try

Semaphore (int capacity, Cyclic barrier)

acquire()

release()

tryAcquire()

CyclicBarrier (int n)

await()

CountDownLatch (int n)

acquires locks ordering 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 objectToExchange)** has to be called by both  
threads and blocks until both have called it

Executor Abstract from Thread Creators  
void execute(Runnable runnable)

Page 16

## ExecutorService

Subinterface of Executor

Further lifecycle management logic

submit<T> 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 completion  
get Blocks until thread finished and Future contains the value  
get<TimeOut, TimeUnit> returns the value  
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 Join Pool on it, ForkJoinTasks can be executed  
↳ parallelStream() → parallelTask

• RecursiveTask<T> • each() ForkSubtask.execute() Create in place JoinTask  
Any Java Collection can be treated as a stream by calling  
the Stream() method  
Various 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  
& and operator or by starting a new requires / ensures line

\existsexists declaration; range-expression; body-expression

\max, \min, \sum, etc

Class Invariants must hold in all user-visible states  
invariant pure keine Substitution

Prinzip 17

- Liskov Substitution Principle for pre- and postconditions of overriding methods
  - Preconditions must ~~be~~ not be more restrictive than <sup>the other</sup> ~~overwritten~~ method Precondition<sub>super</sub>  $\Rightarrow$  Precondition<sub>sub</sub>
  - Postconditions must be at least as restrictive as those of the overwritten methods: Postcondition<sub>sub</sub>  $\Rightarrow$  Postcondition<sub>super</sub>
- Regarding a complete class
  - $\rightarrow$  Pre- and postconditions relations must hold for all methods as stated above
  - $\rightarrow$  The class invariants must be at least as restrictive as those of the superclass invariants<sub>super</sub>  $\Rightarrow$  invariants<sub>super</sub>

## Compiler

Ebenen der Interpretation bzw. Übersetzung

- ~~Übersetzung in Maschinencode (LL(1)T)~~

- Vollständige Übersetzung: Übersetzung in Maschinencode (C/C++)

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

\* Interpretator nach Übersetzung: Analyse der Quelle und Transformation in eine für den Interpretierer geeignete Form (Bsp: Python, Ruby,...)

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

Lexikalische Analyse:

Eingabe: Sequenz von Zeichen

Aufgabe:

Erkennen bedeutungstragende Zeichengruppen: Tokens

• Übergangsweise unbedeutende Zeichen (Pfeilzeichen, Kommentare)

• Bereichsidentifizieren und Zusammenfassen in Struktabelle

Syntaktische Analyse

Eingabe: Sequenz von Tokens

Aufgabe:

Überprüfen, 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

• Typenanalyse: Bezeichner und prüfen Typen von Variablen, Funktionen

• Konsistenzprüfung - alle Einschränkungen des Programmiersprachen eingehalten

Ausgabe: erweitelter Syntaxbaum

Originalprogramme werden später in syntaktisch semantisch korrekt abgebildet.

Zwischencodegenerator, Optimierer  
Autobasis  
Prinzip: Code in sprach- und zielunabhängige ZwischenSprache  
Optimiere Code  
Konstantenfaltung - Bsp: Ersetze  $3+8$  durch  $8$   
Kopienfachschaltung  
Code Verdichtung  
Generische Funktionen extrahieren  
Inlining

## Codegenerierung

Eingabe: abstrakter Syntaxbaum oder Zwischencode  
Auszabe: Erzeugte Code für Zielmaschine  
Anpassung an Konventionen des Zielkreditsystems  
Callsignatur - welche Befehle das Zielsystems benötigen?  
Scheduling - Reihenfolge der Befehle festlegen  
Registerallokation - welche Zwischenergebnisse in Prozessorregistern halten  
Nachoptimierung  
Auszabe: Programm in Assembly oder Maschinencode  
durch Assembler und Linker

## Syntaktische Analyse

Syntaktisch Syntaktische Analyse (Parsing)

Verwendung von Kontextfreier Grammatik ~~um weiter zu gehen~~ i.S. 38%

Ableitungs- Bzr Syntaktbaum

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

AB Abstrakter Syntaxbaum enthält

- nur spezifizierte Token (z.B. Id, Float) als Blätter;
- keine Schlüsselworte als Operatoren
- Operatoren an ihren Knoten
- keine Klammern, sondern Operator-Prioritäten durch Baumstruktur
- keine Ketteproduktionen

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

LL fügt dann LR unten auf, Behandelt = SLL Rekursiv Abhängig, unabhängig von LL

Linksableitig: linke Teilbörne vor rechten

Pro Symbol der Grammatik eine Funktion, jede:  
• passt & konsumiert einen Teil der Eingabe  
• entscheidet anhand nächster Token welche Produktion zu einer passt  
• delegiert an Parse-Funktionen andere Symbole

Rekursiver Rüstung muss Produktion mit h-Lockahead unterscheiden können

$h$ -Anfang  $h : \Sigma$

Für  $X \in \Sigma^*$  ist der  $h$ -Anfang  $h : \Sigma$  des Präfixes der Länge  $h$  von  $X \# \dots$ , wobei  $\# \notin \Sigma$  das Ende des Terminalstrings kennzeichnet.

Für  $X \in (\Sigma \cup V)^+$  ist First<sub>h</sub> definiert als  
First<sub>-Menge</sub>

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

First<sub>h</sub>(X) sind die  $h$ -Anfänge der strings, die aus X generiert werden können.

First und Follow<sub>h</sub>

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

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

Follow<sub>h</sub>(X) =  $\{\beta \mid \exists \alpha, w \in (V \cup \Sigma)^* \text{ mit } s \Rightarrow^* \alpha X w \text{ und } \beta \in \text{First}_h(w)\}$

Follow<sub>h</sub>

Follow<sub>h</sub>(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<sub>h</sub>(α) Follow<sub>h</sub>(A)

Eine (kontextfreie) Grammatik ist genau dann eine 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  $h$ -Lookhead in rekursiven Abstieg passbar

Linksrekurrenz: Verzage  $X \rightarrow Y \alpha \mid Y \beta$ , verzage Entscheidung zw.  $\alpha$  und  $\beta$  bis nachdem  $\gamma$  konsumiert wurde

$$\begin{aligned} X &\rightarrow Y^* X \\ X &\rightarrow \alpha \mid \beta \end{aligned}$$

Linkssehbare kontextfreie Grammatiken sind für kein  $h$  SLL(h)

Faustregeln Grammatikengineering

- Ein Nichtterminal pro Prioritätsstufe
- Häufig Schichtung: Produktionen von niedrig zu hoch Priorität
- q.d.h. dasselbe Nichtterminal zweimal auf der Rechten Seite
- Assoziativität von Operatoren: Wie ist  $a+b+c$  zu passen?  
 $(a+b)+c \Rightarrow$  Linkspriorisierung  $E \rightarrow E + T$   
 $a+(b+c) \Rightarrow$  Rechtspriorisierung  $E \rightarrow T+E$

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

Grammatik eindeutig  $\Rightarrow$  Assoziativität durch grammatisch vorgegebene

Implementierung von ASTs

zu jedem syntaktischen Kategorien eine Klasse

zu jeder Blattkategorie jeder Kategorie in Kategorienstruktur

Rekursives Abstieg: Linkssozialitität s.S. 414

Progr 2d

## Semantische Analyse

Namen in SIMPLE und Java:

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

Implementierung der Namensanalyse:

Vorgehen: Durchlaufe AST und sammle 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 merke vorher Definition in Liste
  - Beim Verlassen eines Namensbereichs stelle vorherige Definitionen anhand der Liste wieder her

AST - Transversierung  
Nutze visitor pattern

Java Bytecode

Virtuelle Maschine

Heap

Method Area: Code für Methoden nutzbar

Runtime Constant Pool

Threads: Da Thread:

Program Counter

JVM State

Native Method Stack: Für Laufzeitssystem

Stack basierter Bytecode: Spezialisierung Rücksicht auf Operandenstack

Instruktionen

explizit für direkt add(int), add(float)

Instruktionsarten

- Lesen/Schreiben von lokalen Variablen (Load, Store etc.)
- Lesen/Schreiben von Feldern (getfield, putfield, ...)
- Sprungbefehle (ifeq, ifne, tableswitch, ...)
- Methodenauffüllung (invocation, invokestatic, ...)
- Objektzugriff (new, newarray, ...)
- Arithmetische Berechnungen (lmul, ladd, ...)

## Methodenausdrücke

1. this-Parameter pa auf den Operandenstack (falls nicht static)
2. Parameter p1...pn auf Operandenstack
3. rückwärtslaufend / in reverse stack ausführen. Automatisch passiert
  1. lokale Aktivations Record erlegen
  2. Rückgradaresse (Program Counter + 1) auf und dynamischen Vorrangspfeil (Framepointer) in Activation Record sichern
  3. Neuen Framepointer setzen
  4. Parameter (pa, ..., pn, zgl. pa) von Operandenstack in neuen Activation Record kopieren
  5. Zu Methodenaufruf springen
4. Block wird ausgelöscht
5. Rückgraderest auf den Operandenstack
6. Ausführen ausblöhen (ireturn, retur ..., ...) automatisch passiert
7. Alten Framepointer setzen und zur Rückgradaresse springen
8. Rückgraderest aus Operandenstack des neuen Activation Block in alten Operandenstack hinzun.

## Descriptoren

- Namen von Typenfeldern und Methoden muss festgelegtes Schema entsprechen
- Objekttypen `java.lang.Object` → Ljava/lang/Object;
  - primitive Typen int → I, void → V, boolean → Z, ...
  - Methoden `void foo(int, Object)` → foo(ILjava/lang/Object;)V
  - Descriptors: (Parametertypen) Rückgabefall, Identifikator des Name/Descriptor
  - Fields: boolean b → B; Identifikator nur des "Name"
  - Konstruktoren: Name ist <init>, statische Initializer <CLinit>

## Objekt erzeugen &amp; initialisieren

1. Objekt anlegen → Speicher reservieren
2. Objekt initialisieren → Konstruktor aufrufen

Umkehrte polare Notation (CUPN) zuerst öffnen, dann ausführbare Operation

7\*4 in CUPN 74\*  
 2\*(2+3) : UIN 223+\*  
 S+9+3\*5 : UPN 59+35\*+

## Bytecodeerzeugung für arith. Ausdrücke

Absturzter Syntaxbaum

Postfixordnung für tieferende erzeugt UIN

Postfixordnung. Befehlserzeugung beim Verlassen eines Knotens (absonderum CUPN-Reihenfolge integriert Bytecode-Befehle sequentiell nicht gestört sind)

## Kontrollfluss

- Jeder AST-Knoten ruft rekursiv `codeGen()` Methode seiner Kindheit
- Jede Kontrollflussknoten generiert ihren Code so, dass sie nur über `fall through` verlassen wird
- ⇒ Hier kommt ausgelöste codeGen() Methoden erzeugen notwendig ausgelöster Code
- Code erzeugt bei Bedingungen (z.B. `L=`) schaut zw. Sprungübersetzung zu dem dann zu erzeugte Code je nach Auswertung der Bedingung sprangt.

W. Konditionierung (11.64) Erzeuge falls 1 das angegeben wird, falls reelle Werte ausgewählt werden muss

Negation: Nein Zusätzlicher Beikasten sondern Veranschlagter Sprungzettel  
Propri 22

# 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) -> val

? add

? sub

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

ifeq label Branch zu label, wenn Stackwert 0 (val ->)

ifnull

ifle

null (val ->)

≤ 0 (val ->)

getfield index Beliebige field value by index im const pool (0 get -> value)  
putfield index Setze

const auf 2.

Stack value

(objectref, value ->)

New index Neues Objekt von index im Const pool (-> 0 get)  
New array atype Neues Array von Typ aType (Count -> arrayref)

goto label

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

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

instanceof

invokenv rufe virtuelle Methode auf, ergibt sich ab Stack, index ab const pool

static

statisch

special

instance

if\_icmpne

v1 = > v2

if\_icmpge

≥

gt

>

le

≤

lt

<

ne

≠