Informatik III - Zusamenfassung

Patrick Pletscher

15. September 2004

1 Einführung in Haskell

1.1 Grundbegriffe der funktionalen Programmierung

- Funktionen und Werte
  - Funktionen berechnen Werte
  - Funktionen können selbst als Werte betrachtet werden
- Keine Seiteneffekte: f(x) ergibt immer das gleiche Ergebnis

1.2 Syntax

ggT als funktionales Programm

```
ggt x y
  | x == y    = x
  | y > x     = ggt y x
  | otherwise = ggt (x-y) y
```

Anstatt anzugeben, *wie* etwas berechnet wird, gibt man an, *was* berechnet werden soll.

Das Programm besteht aus mehreren Fällen:

```
Name x1 ... xn
   | guard1 = fall1
     :
   | guradm = fallm
```

Typen

Entweder der User gibt Typen mit Funktions-Definitionen an, z.B.

```
ggt :: Int -> Int -> Int
```

oder das System berechnet die Typen selbst.

Typ Int

- Beschränkt auf 32bit
- Funktionen sind meist Präfixform, also:

```
? mod 7 2
```

Man kann das Ganze aber auch Infix schreiben:

```
? 7 'mod' 2
```

Eine Infix Binary-Funktion heisst "Operator".

Typ Bool

- Werte: True, False
- Operatoren (infix) `&&, ||, not`

Typ Tupel

Beispiel: Student hat Name, Martikelnummer, Anfangsjahr

Rekord-Typ:	`(String, Int, Int)`
mit Element:	`("Fritz", 1234, 2002)`

Muster und Funktionsdefinition

Funktionsdefinition mit Muster `mi` und Guards `gi` kann kombiniert werden:

```
fun m1 m2 ... mn
  | g1   = e1
    :
  | gm   = em
  | otherwise = e  ---optional!
```

Lokale Reichweite mit let und where

- `let`

```
let x1 = e1
    :
    xn = en
in e
```

`let` baut einen Ausdruck von anderen aus:

- `xi` kann sowohl Variable oder Funktionen (lokal) binden
- Eine Definition kann auch andere benutzen

Beispiel

```
f x = let sq y = y * y
          in sq x + sq x
```

Wir können f auswerten, aber nicht sq.

- `Where`

```
f p1 p2 .. pk
   | g1 = e1
   | g2 = e2
     :
   | gk = ek
  where
    v1 a1 ... an = r1
    v2 = r2
    :
```

  - `where` wird direkt nach einer Funktionsdefinition gegeben
  - Man kann Bindings über mehrere Guards definieren

Beispiel

```
maxThreeOccurs n m p = (maxVal, eqCount)
 where maxVal = maxiThree n m p
       eqCount = equalCount maxVal n m p

maxiThree a b c = max a (max b c)

equalCount val n m p = isN + isM + isP
 where isN = if n == val then 1 else 0
       isM = if m == val then 1 else 0
       isP = if p == val then 1 else 0
```

1.3 Korrektheit

Terminierung

Falls die Funktion f durch andere Funktionen $g_1, \ldots, g_k$ definiert wird und diese g_i terminieren, dann terminiert f.

Das Problem sind *Rekursionen*:
Hinreichende Terminationsbedingung: Argumente werden entlang einer *wohlfundierten* Ordnung kleiner. Wobei eine Ordnung $>$ wohlfundiert ist, wenn es keine unendlich absteigende Ketten gibt $x_1 > x_2 > x_3 > \ldots$. Beispiel: $> \mathbb{N}$, Gegenbeispiele: $> \mathbb{Z}, > \mathbb{R}$

Beispiel: Fakultät einer Zahl

```
fac 0 = 1
fac n = n * fac (n-1)
```

`fac n` hat nur `fac (n-1)` als rekursiven Aufruf und $n > n-1$, wobei $>$ die Ordnung über den natürlichen Zahlen ist.

Korrektheit - Richtiges Verhalten

Basiert auf einfacher Idee: Funktionen sind Gleichungen, man kann also mathematische Methoden und auch logische Kalküle benutzen, so z.Bsp.:

- "Excluded Middle": für alle Aussagen P gilt immer:
$$P \vee \not P$$
- Fallunterscheidung: Gegeben sei $Q \vee R$. Für P müssen wir zeigen:
  1. Aus der Annahme Q folgt P und
  2. Aus der Annahme R folgt P
- Induktion.

2 Listen

2.1 Listentypen

Listentyp, aufgebaut mit Listentyp-Konstruktor: Wenn T ein Typ ist, dann ist $[T]$ ein Typ.

Elemente von $[T]$:

- Leere Liste $[] :: [T]$
- Wenn $a :: T$ und $l :: [T]$, dann $(a : l) :: [T]$

Abkürzung: $1 : (2 : (3 : []))$ wird als $[1, 2, 3]$ geschrieben.

Weitere Abkürzungen:

```
? [3..6]
[3,4,5,6]::[Int]

? [6..3]
[]::[Int]
```

`[n,p..m]` heisst Zahlen von n zu m in Schritten $p-n$:

```
? [7,6..3]
[7,6,5,4,3]::[Int]

? [0.0, 0.3 .. 1.0]
[0.0, 0.3, 0.6, 0.9] :: [Double]
```

2.2 Standardfunktion über Listen

length

```
length [] = 0
length (a:x) = 1 + length x
```

Append (nicht nur für Strings)

```
[] ++ y = y
(a:x) ++ y = a:(x++y)
```

2.3 Muster

Ein Muster wird wie folgt induktiv definiert:

Konstante

Variable

Tupel : $(p_1, \ldots, p_k)$, wobei auch p_i Muster sind

Liste : $(p_1 : p_2)$ wobei auch p_i Muster sind

'Wild cards' : _

Aber ein Muster muss *linear* sein: jede Variable darf nur maximal ein Mal vorkommen.

2.4 Musteranpassung

Argument a wird einem Muster p erfolgreich angepasst gdw p ist:

Konstante : $a = p$

Variable x: erfolgt immer mit Zuordnung $x = a$

Tupel $(p_1, \ldots, p_k)$: $a = (a_1, \ldots, a_k)$ und a_i an p_i angepasst wird.

Liste $(p_1 : p_2)$: a ist eine nicht leere Liste und p_1 wird dem Kopf von a angepasst und p_2 wird dem Rest von a angepasst.

'Wild cards' : immer, aber keine Bedingung (fungiert nur als Test)

```
zip (a:x) (b:y) = (a,b) : zip x y
zip _ _ = []
```

Insertion Sort

```
isort :: [Int] -> [Int]
isort [] = []
isort (a:x) = ins a (isort x)

ins :: Int -> [Int] -> [Int]
ins a [] = [a]
ins a (b:y)
 | a <= b = a:(b:y)
 | otherwise = b:(ins a y)
```

2.5 Listen-Komprehension

Notation für sequenzielle Verarbeitung der Listen, welche mit Tests kombiniert werden kann.

```
[2*a | a <- l, b1(a), ..]
```

Quicksort

```
q [] = []
q (a:x) = q [y | y<-x, y <=a] ++
  [a] ++ q[y |y<-x, y>a]
```

3 Abstraktion

3.1 Funktionen höherer Stufe

Argumente dürfen selber Funktionen sein.

```
(Int -> Int) -> [Int] -> [Int]
```

map

```
-- higher order (function f is an argument)
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (a:x) = f a : map f x
```

Ähnlichkeit mit Listenkomprehension:

```
map f l = [f a | a <- l]
```

foldr

rechts-assoziatives Fold:

$$foldr(\oplus)\ e\ [l_1, \ldots, l_n] = l_1 \oplus (l_2, \ldots \oplus (l_n \oplus e))$$

```
foldr :: (a -> b -> b) ->  b -> [a] -> b
foldr f e [] = e
foldr f e (x:xs) = f x (foldr f e xs)
```

Damit kann man z.Bsp. concat definieren:

```
concat xs = foldr (++) [] xs
? concat [[1,2,3],[4],[5,6]]
[1,2,3,4,5,6] :: [Int]
```

foldl

links-assoziatives Fold:

$$foldl(\oplus)\ e\ [l_1, \ldots, l_n] = ((e \oplus l_1) \oplus l_2) \ldots \oplus l_n$$

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f e [] = e
foldl f e (x:xs) = foldl f (f e x) xs
```

3.2 Typen und Polymorphismus

Polymorphe Typen enthalten Typvariablen:

```
length :: [t] -> Int
```

Definition: Ein Typ w für f ist ein *allgemeiner* (auch *prinzipaler*) Typ gdw. für alle Typen s für f, s eine Instanz von w ist.

3.3 Funktionen als Werte

Funktionen können als Werte geliefert werden. So z.Bsp. Funktionskomposition:

$$(f \circ g)x = f(gx)$$

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f (g x)
```

3.4 λ-Ausdrücke

Beispiel:

```
? map (\x -> x * 2) [2,3,4]
[4,6,8] :: Int

? foldr (\x y -> x * y) 1 [1,2,3,4]
24 :: Int
```

3.5 Partielle Anwendung

Jede Funktion nimmt genau ein Argument
`multiply :: Int -> Int -> Int` bedeutet
`multiply :: Int -> (Int -> Int)`

`(multiply 2)` ist z.Bsp. eine Funktion, die ein Argument nimmt und als Resultat die Verdoppelung ist.

3.6 Mehrere Argumente vs. Tupeln

```
f :: (Int,Int) -> Int
f (x,y) = x * y + 17

g :: Int -> Int -> Int
g x y = x * y +17
```

Tupel-Argumente: keine partielle Auswertung.

Aber äquivalent im folgenden Sinn:

```
curry :: ((a,b) -> c) -> a -> b -> c
uncurry :: a -> b -> c -> (a,b) -> c

curry f    = f' where f' x1 x2 = f (x1,x2)
uncurry f' = f  where f(x1,x2) = f' x1 x2
```

Beispiele

```
? f (3,4)
29 :: Int

? curry f 3 4
29 :: Int
```

4 Programmierung höherer Ordnung

4.1 Filter

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (h:t)
  | p h = h:rest
  otherwise = rest
              where rest = filter p t
```

4.2 Abstrahieren allgemeiner Operationen

Map: Funktionale Anwendung auf jedes Element.
Filter: Selektion

Fold: Kombination von Elementen

4.3 Map und Filter versus Komprehension

Map und Filter können mit Komprehension implementiert werden

```
map f l = [f x | x <- l]
filter p l = [x | x <- l, p x]
```

Auch die Umkehrung gilt auch: `[e | p <- s]`

```
let fn p = e in map fn s
```

Guards erfordern Filter: `[e | p <- s, g]` übersetzt als

```
let fn p = e
    pred p = g
in map fn (filter pred s)
```

4.4 zipWith

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f _ _ = []
```

5 Typklassen und Polymorphismus

5.1 Typklassen

```
allEqual :: Eq a => a -> a -> a -> Bool
allEqual x y z = (x == y) && (y == z)
```

Funktioniert für genau die Typen a, die zur Klasse Eq gehören.

Eq ist die Gleichheitsklasse, eine Klasse definiert eine Menge von Typen.

Definition der Eq Klasse

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x /= y      = not (x==y)
```

Eine Definition beeinhaltet:

Klassenname: Eq

Signatur: Liste von Namen und Typen

(Optional Standard-) Definition: kann später überschrieben werden

Elemente von Klassen heissen *Instanzen.*

Instanzen

Durch eine Definition von Signaturfunktionen wird ein Typ in einer Klasse instanziert.

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

Ageleitete Klassen

```
class Eq a => Ord a where
  (<), (>), (<=), (>=) :: a -> a -> Bool
  max, min :: a -> a -> a

  x < y = (x <= y && x /= y)
  x >= y = y <= x
  x > y = y <= x && x/= y

  max x y | x <= y    = y
          | otherwise = x
  min x y | x <= y    = x
          | otherwise = y
```

Wenn a zu Ord gehört, dann muss a auch zu Eq gehören. Funktionen für Eq werden vererbt, einige neue müssen gegeben werde:

```
Instance Ord Int where (<=) = primeLeInt
```

5.2 Typisierung

Ziel von Typisierung:

- schnell entscheidbare statische Analyse
- erlaubt so viel Allgemeinheit und Wiederverwendbarkeit wie möglich.
- Keine Laufzeitfehler: Subjektreduktion

  Sei $e \hookrightarrow e'$ und $\vdash e :: \tau$. Dann gilt auch $\vdash e' :: \tau$

Mini-Haskell - Syntax

Programme sind Terme (Sie Variablen $\mathcal{V}$ und Zahlen $\mathcal{Z}$ gegeben)

$$\begin{aligned} t \quad ::= \quad & \mathcal{V} \mid \lambda x.t \mid (t_1 t_2) \mid \\ & \mathit{true} \mid \mathit{false} \mid \mathbf{iszero}(t) \mid \\ & \mathcal{Z} \mid t_1 + t_2 \mid t_1 \times t_2 \mid \mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \mid \\ & (t_1, t_2) \mid \mathbf{fst}(t) \mid \mathbf{snd}(t) \end{aligned}$$

Typisierung

Typen ($\mathcal{V}_\mathcal{T}$ ist die Menge von Typ Variablen, $a, b, \ldots$)

$$\tau ::= \mathcal{V}_\mathcal{T} \mid \mathit{Bool} \mid \mathit{Int} \mid (\tau, \tau) \mid \tau \to \tau$$

Notation des Typsystems basiert auf Typurteilen $A \vdash e :: \tau$

- A ist 'Symboltabelle': Abbildung von Identifikatoren auf Typen
- e ist ein Ausdruck
- τ ist ein Typ

Regeln für Kern λ-Kalkül

- Symboltabelle Axiom: $\ldots, x : \tau, \ldots \vdash x :: \tau$
- Abstraktion ($x \notin A$):

$$\frac{A, x : \sigma \vdash t :: \tau}{A \vdash \lambda x.t :: \sigma \to \tau}$$

- Applikation:

$$\frac{A \vdash t_1 :: \sigma \to \tau \qquad A \vdash t_2 :: \sigma}{A \vdash (t_1 t_2) :: \tau}$$

  Die Applikation vereingt sich links. Bsp. xyz steht für $(xy)z$

Weitere Typregeln für mini-Haskell

- Basistypen

$$A \vdash n :: \mathit{Int}$$
$$A \vdash \mathit{true} :: \mathit{Bool}$$
$$A \vdash \mathit{false} :: \mathit{Bool}$$

- Operationen ($\mathbf{op} \in \{+, \times\}$)

$$\frac{A \vdash t :: \mathit{Int}}{A \vdash \mathbf{iszero}(t) :: \mathit{Bool}}$$

$$\frac{A \vdash t_1 :: \mathit{Int} \qquad A \vdash t_2 :: \mathit{Int}}{A \vdash t_1\ \mathbf{op}\ t_2 :: \mathit{Int}}$$

$$\frac{A \vdash t_0 :: \mathit{Bool} \qquad A \vdash t_1 :: \tau \qquad A \vdash t_2 :: \tau}{A \vdash \mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 :: \tau}$$

- Tupeln

$$\frac{A \vdash t_1 :: \tau_1 \qquad A \vdash t_2 :: \tau_2}{A \vdash (t_1, t_2) :: (\tau_1, \tau_2)}$$

$$\frac{A \vdash t :: (\tau_1, \tau_2)}{A \vdash \mathbf{fst}(t) :: \tau_1}$$

$$\frac{A \vdash t :: (\tau_1, \tau_2)}{A \vdash \mathbf{snd}(t) :: \tau_2}$$

6 (Algebraische) Datentypen

6.1 Algebraische Datentypen

Anstatt gegebene Typen zu verwenden, kann ein Typ deklariert werden, der die Objekte 'direkt' modelliert.

Aufzählungstypen (Verbunde)

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

Syntax:

- fängt mit Keyword `data` an
- gibt verschiedene eindeutig genannte Konstruktoren
- erster Buchstabe von Konstruktoren muss gross sein

Produkttypen

```
data People = Person Name Age
type Name = String
type Age = Int
```

Aufzählungs- und Produkttypen

Kombinationen möglich

```
data Shape = Circle Float |
             Rectangle Float Float
```

- Konstruktoren dienen als Funktionen

```
? Rectangle
<<function>> :: Float -> Float -> Shape
```

- Beispiele von Fuktionen:

```
area :: Shape -> Float
area (Circle r) = pi * r * r
area (Rectangle h w) = h * w
```

Integration mit Klassen

Klasseninstanz kann explizit kreiert werden:

```
data Foo = D1 | D2 | D3

instance Eq Foo where
  D1 == D1 = True
  D2 == D2 = True
  D3 == D3 = True
  _  == _  = False
```

In einigen Fällen können Klasseninstanzen automatisch *abgeleitet* werden.

```
data Foo = D1 | D2 | D3
           deriving (Eq, Ord, Enum, Show)
```

6.2 Allgemeine Definition

$$
\begin{aligned}
data\ T \quad = \quad & Con_1 T_{11} \dots T_{1k_1} \\
& | Con_1 T_{21} \dots T_{2k_2} \\
& \vdots \\
& Con_n T_{n1} \dots T_{nk_n}
\end{aligned}
$$

T_{ij} sind andere Typen, möglicherweise auch T (rekursiv).

Rekursivtypen

```
data Expr = Lit Int | Add Expr Expr |
            Sub Expr Expr
```

6.3 Bäume

```
data Tree t = Leaf | Node t (Tree t) (Tree t)
            deriving (Eq, Ord, Show)
```

6.4 Programmierung höherer Ordnung mit Datentypen

Vorgehensweise vieler Funktionen höherer Ordnung verallgemeinbar

```
mapTree :: (t -> u) -> Tree t -> Tree u
mapTree f Leaf = Leaf
mapTree f (Node x t1 t2) =
    Node (f x) (mapTree f t1) (mapTree f t2)
```

6.5 treeFold

```
treeFold :: (a -> b -> b -> b) -> b -> Tree a -> b

treeFold f e Leaf = e
treeFold f e (Node x l r) =
  f x (treeFold f e l) (treeFold f e r)
```

Von Bäumen zu Listen

```
preorder t = treeFold (\x y z -> [x] ++ y ++ z) [] t

inorder t = treeFold (\x y z -> y ++ [x] ++ z) [] t

postorder t = treeFold (\x y z -> y++ z ++ [x]) [] t
```

7 Programmierung mit verzögerter Auswertung

Haskell basiert auf verzögerter Auswertung: Ein Argument wird nur ausgeführt, wenn es gebraucht wird.

Mögliches Problem: Duplizierte Berechnung, z.B. `h x = x + x`.

- Dieselben Ausdrücke würden hier zwei Mal ausgewertet.
- Kann vermieden werden: Beide können gleichzeitig reduziert werden.
- Implementierungsidee: Ausdrücke werden als Graphen anstatt als Bäume modelliert.

7.1 Auswertung - Diverses

Funktionen werden von aussen nach innen ausgeführt und sonst von links nach rechts.

8 Substitution

$$
\begin{aligned}
free(x) &= \{x\} \\
free(\lambda x.M) &= free(M) - \{x\} \\
free(MN) &= free(M) \cup free(N)
\end{aligned}
$$

8.1 λ-Kalkül - Substitution

$M[x \leftarrow N]$ bedeutet: x wird durch N in M ersetzt.

1. $x[x \leftarrow N] = N$
2. $y[x \leftarrow N] = y$, falls $y \neq x$
3. $(PQ)[x \leftarrow N] = (P[x \leftarrow N]Q[x \leftarrow N])$
4. $(\lambda x.P)[x \leftarrow N] = \lambda x.P$
5. $(\lambda y.P)[x \leftarrow N] = \lambda y.(P[x \leftarrow N])$, falls $y \neq x$ und $y \notin FV(N)$
6. $(\lambda y.P)[x \leftarrow N] = \lambda z.((P[y \leftarrow z])[x \leftarrow N])$ falls $y \neq x$ und $y \in FV(N)$, wobei $z \notin FV(NP)$

9 Semantik von Programmiersprachen - Übersicht

9.1 Semantik (statisch vs. dynamisch)

Statische Semantik

Was muss zur Compilezeit überprüft werden?

- Typregeln, Typüberprüfung
- statische Analyse (z.B. Definite Assignment)
- Auflösung von Namen
- Auflösung von Methoden (für überladene Methoden)

Dynamische Semantik

Was geschieht zur Laufzeit?

- Ausführung des Programms
- Was definiert den Zustand eines Programms?
- Was ist der Effekt eines Programms auf den Zustand?

9.2 Dynamische Semantik

Denationelle Semantik

Programme = mathematische Objekte

- Ein Programm wird betrachtet als (partielle) Funktion in einem mathematischen Raum.
- Gut für deklarative Sprachen (funktionale Programmierung).
- Zu kompliziert für objekt-orientierte Sprachen mit Vererbung, Exception Handling, Threads.

Operationelle Semantik

Varianten:

- Natürliche Semantik (Big-Step Semantics)
- SOS (Structural Operational Semantics)
- ASMs (Gurevich Abstract State Machines)

9.3 Eine einfache imperative Programmiersprache

Nur zwei Typen: `boolean, integer`

Bedeutung der Anweisungen (informale Semantik)

`skip`	Mache nichts (leere Anweisung)
x`:=` e	Weise x den Wert von e zu.
s_1 ; s_2	Führe zuerst s_1 aus, danach s_2
`if` b `then` s_1 `else` s_2 `end`	Falls b wahr ist, führe s_1 aus, sont s_2.
`while` b `do` s `end`	Solange b wahr ist, führe s aus.
`var` x `:=` e `in` s `end`	Erzeuge eine neue lokale Variable x mit dem Wert e und führe s aus.
$p(\vec{e}, \vec{z})$	Rufe die Prozedur p mit den Werten von $\vec{e}$ und den Variablen $\vec{z}$ auf.

Bedingung: Die Variablen in $\vec{z}$ müssen paarweise verschieden sein.

10 Natürliche Semantik von Programmen

Idee: Zu einem Programm π definiert man induktiv Tripel

$$\sigma \multimap [s] \rightarrowtail \sigma'$$

mit der Bedeutung:
Die Ausführung der Anweisung s terminiert für π und führt den Anfangszustand σ über in den Endzustand σ'.

10.1 Regeln der natürlichen Semantik

Die natürliche Semantik eines Programms wird durch Regeln

$$\frac{\varphi_1 \dots \varphi_n}{\psi}\ Condition$$

Dabei sind $\varphi_1, \dots, \varphi_n$ und ψ Tripel

$$\sigma \multimap [s] \rightarrowtail \sigma'$$

wobei $\sigma, \sigma' \in State$ und $s \in Stm$.

Bedeutung der Regel:
Falls $Condition$ und $\varphi_1, \dots, \varphi_n$, dann ψ.

10.2 Natürliche Semantik

$$\frac{}{\sigma \multimap[\texttt{skip}]\rightarrowtail \sigma}$$

$$\frac{}{\sigma \multimap[x\,\texttt{:=}e]\rightarrowtail \sigma[x \mapsto \mathcal{A}[e]\sigma]}$$

$$\frac{\sigma \multimap[s_1]\rightarrowtail \sigma' \qquad \sigma' \multimap[s_2]\rightarrowtail \sigma''}{\sigma \multimap[s_1; s_2]\rightarrowtail \sigma''}$$

$$\frac{\sigma \multimap[s_1]\rightarrowtail \sigma'}{\sigma \multimap[\texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}]\rightarrowtail \sigma'} \text{ falls } \mathcal{B}[b]\sigma = 1$$

$$\frac{\sigma \multimap[s_2]\rightarrowtail \sigma'}{\sigma \multimap[\texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}]\rightarrowtail \sigma'} \text{ falls } \mathcal{B}[b]\sigma = 0$$

$$\frac{\sigma \multimap[s]\rightarrowtail \sigma' \qquad \sigma' \multimap[\texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}]\rightarrowtail \sigma''}{\sigma \multimap[\texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}]\rightarrowtail \sigma''} \text{ falls } \mathcal{B}[b]\sigma = 1$$

$$\frac{}{\sigma \multimap[\texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}]\rightarrowtail \sigma} \text{ falls } \mathcal{B}[b]\sigma = 0$$

$$\frac{\sigma[x \mapsto \mathcal{A}[e]\sigma] \multimap[s]\rightarrowtail \sigma'}{\sigma \multimap[\texttt{var}\ x\ \texttt{:=}\ e\ \texttt{in}\ s\ \text{end}]\rightarrowtail \sigma'[x \mapsto \sigma(x)]} \text{ falls } x \in \mathsf{dom}(\sigma)$$

$$\frac{\sigma[x \mapsto \mathcal{A}[e]\sigma] \multimap[s]\rightarrowtail \sigma'}{\sigma \multimap[\texttt{var}\ x\ \texttt{:=}\ e\ \texttt{in}\ s\ \texttt{end}]\rightarrowtail \sigma'\backslash\{x \mapsto \sigma'(x)\}} \text{ falls } x \notin \mathsf{dom}(\sigma)$$

$$\frac{\{\vec{x} \mapsto \mathcal{A}[\vec{e}]\sigma, \vec{y} \mapsto \sigma(\vec{z})\} \multimap[s]\rightarrowtail \sigma'}{\sigma \multimap[p(\vec{e};\vec{z})]\rightarrowtail \sigma[\vec{z} \mapsto \sigma'(\vec{y})]}$$

für $\texttt{procedure}\ p(\vec{x};\vec{y})\ \texttt{begin}\ s\ \texttt{end}$

10.3 Die zugewiesenen Variablen

$$\mathrm{AV}(\texttt{skip}) = \emptyset$$
$$\mathrm{AV}(x \texttt{ := } e) = \{x\}$$
$$\mathrm{AV}(s_1 \texttt{; } s_2) = \mathrm{AV}(s_1) \cup \mathrm{AV}(s_2)$$
$$\mathrm{AV}(\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ end}) = \mathrm{AV}(s_1) \cup \mathrm{AV}(s_2)$$
$$\mathrm{AV}(\texttt{while } b \texttt{ do } s \texttt{ end}) = \mathrm{AV}(s)$$
$$\mathrm{AV}(\texttt{var } x \texttt{ := } e \texttt{ in } s \texttt{ end}) = \mathrm{AV}(s) \backslash \{x\}$$
$$\mathrm{AV}(p(\vec{e}; \vec{z})) = \{\vec{z}\}$$

Die Menge $\mathrm{AV}(s)$ ist eine Obermenge der Variablen, die von der Anweisung s zur Laufzeit tatsächlich geändert werden.

10.4 Technische Details

Die natürliche Semantik erhält den Definitionsbereich der Zustände.

Lemma 10.1. *Falls $\sigma \text{ -}[s]\rightarrow \sigma'$, dann ist $\mathsf{dom}(\sigma) = \mathsf{dom}(\sigma')$.*

Die natürliche Semantik von deterministischen Programmen ist deterministisch.

Lemma 10.2. *Falls $\sigma \text{ -}[s]\rightarrow \sigma'$ und $\sigma \text{ -}[s]\rightarrow \sigma''$, dann ist $\sigma' = \sigma''$.*

Nur zugewiesene Variablen werden geändert.

Lemma 10.3. *Falls $\sigma \text{ -}[s]\rightarrow \sigma'$ und $x \in \mathsf{dom}(\sigma)\backslash AV(s)$, dann ist $\sigma'(x) = \sigma(x)$.*

10.5 Koinzidenz

Lemma 10.4. *Falls σ und τ auf den Variablen von e übereinstimmen, dann ist $\mathcal{A}[e]\sigma = \mathcal{A}[e]\tau$.*

Lemma 10.5. *Falls σ und τ auf den Variablen von b übereinstimmen, dann ist $\mathcal{B}[b]\sigma = \mathcal{B}[b]\tau$.*

Lemma 10.6. *Sei V eine Menge von Variablen, welche die freien Variablen von s umfasst. Falls σ und τ auf V übereinstimmen und $\sigma \text{ -}[s]\rightarrow \sigma'$, dann gibt es ein τ' so, dass $\tau \text{ -}[s]\rightarrow \tau'$ und τ und τ' auf V übereinstimmen.*

10.6 Semantische Äquivalenz von Programmen

Definition 10.1. Zwei Anweisungen s_1 und s_2 sind *semantsich äquivalent* (geschrieben: $s_1 \equiv s_2$), falls für alle Zustände σ und σ' gilt:

$$\sigma \text{ -}[s_1]\rightarrow \sigma' \quad \Leftrightarrow \quad \sigma \text{ -}[s_2]\rightarrow \sigma'$$

11 ASM Semantik von Programmen

Idee: Ein Cursor wandert in kleinen Schritten durch den abstrakten Syntaxbaum des Programms und führt dabei Aktionen aus.

11.1 Transitionsregeln der ASM-Semantik

Leere Anweisung

$$^{\blacktriangledown}\texttt{skip} \rightarrow {}^{\blacktriangle}\texttt{skip}$$

Zuweisung

$$^{\blacktriangledown}(x \texttt{ := } e) \rightarrow {}^{\blacktriangle}(x \texttt{ := } e)$$
$$\mathsf{store}(\mathsf{env}(x)) := \mathcal{A}[e](\mathsf{store} \circ \mathsf{env})$$

Sequenzielle Komposition

$$^{\blacktriangledown}(s_1 \texttt{ ; } s_2) \rightarrow {}^{\blacktriangledown}s_1$$
$$({}^{\blacktriangle}s_1 \texttt{ ; } s_2) \rightarrow {}^{\blacktriangledown}s_2$$
$$(s_1 \texttt{ ; } {}^{\blacktriangle}s_2) \rightarrow {}^{\blacktriangle}(s_1 \texttt{ ; } s_2)$$

If-Then-Else-Anweisung

$$^{\blacktriangledown}\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ end} \rightarrow$$
$$\textbf{if } \mathcal{B}[b](\mathsf{store} \circ \mathsf{env}) = 1 \textbf{ then } {}^{\blacktriangledown}s_1 \textbf{ else } {}^{\blacktriangledown}s_2$$

$$\texttt{if } b \texttt{ then } {}^{\blacktriangle}s_1 \texttt{ else } s_2 \texttt{ end} \rightarrow {}^{\blacktriangle}\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ end}$$

$$\texttt{if } b \texttt{ then } s_1 \texttt{ else } {}^{\blacktriangle}s_2 \texttt{ end} \rightarrow {}^{\blacktriangle}\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ end}$$

While-Anweisung

$$^{\blacktriangledown}\texttt{while } b \texttt{ do } s \texttt{ end} \rightarrow$$
$$\textbf{if } \mathcal{B}[b](\mathsf{store} \circ \mathsf{env}) = 1 \textbf{ then } {}^{\blacktriangledown}s \textbf{ else } {}^{\blacktriangle}\texttt{while } b \texttt{ do } s \texttt{ end}$$

$$\texttt{while } b \texttt{ do } {}^{\blacktriangle}s \texttt{ end} \rightarrow {}^{\blacktriangledown}\texttt{while } b \texttt{ do } s \texttt{ end}$$

Lokale Variablendeklaration

$$^{\blacktriangledown}\texttt{var } x := e \texttt{ in } s \texttt{ end} \rightarrow {}^{\blacktriangledown}s$$
$$\textbf{let } \alpha = \textbf{new}(\mathsf{Adr}) \textbf{ in}$$
$$\mathsf{store}(\alpha) := \mathcal{A}[e](\mathsf{store} \circ \mathsf{env})$$
$$\mathsf{env}(x) := \alpha$$

$$\texttt{var } x \texttt{ := } e \texttt{ in } {}^{\blacktriangle}s \texttt{ end} \rightarrow {}^{\blacktriangle}\texttt{var } x \texttt{ := } e \texttt{ in } s \texttt{ end}$$

Prozeduraufruf

$$^{\blacktriangledown}p(\vec{e}; \vec{z}) \rightarrow \texttt{procedure } p(\vec{x}; \vec{y}) \texttt{ begin} {}^{\blacktriangledown}s \texttt{ end}$$
$$\textbf{let } \vec{v} = \mathcal{A}[\vec{e}](\mathsf{store} \circ \mathsf{env}) \textbf{ and } \vec{\alpha} = \textbf{new}(\mathsf{Adr}) \textbf{ in}$$
$$\mathsf{frames} := \mathit{push}(\mathsf{frames}, \langle \mathsf{pos}, \mathsf{env} \rangle)$$
$$\mathsf{store}(\vec{\alpha}) := \vec{v}$$
$$\mathsf{env} := \{\vec{x} \mapsto \vec{\alpha}, \vec{y} \mapsto \mathsf{env}(\vec{z})\}$$

Rückkehr von einem Prozeduraufruf

$$\texttt{procedure } p(\vec{x}; \vec{y}) \texttt{ begin} {}^{\blacktriangle}s \texttt{ end} \rightarrow$$
$$\textbf{let } \langle oldpos, oldenv \rangle = \mathit{top}(\mathsf{frames}) \textbf{ in}$$
$$\mathsf{frames} := \mathit{pop}(\mathsf{frames})$$
$$\mathsf{pos} := oldpos$$
$$\mathsf{env} := oldenv$$
$$\mathsf{mode} := \mathsf{up}$$

11.2 Natürliche Semantik ⇒ ASM-Semantik

Theorem 11.1. *Falls in der natürlichen Semantik $\sigma \text{ -}[s]\rightarrow \sigma'$ herleitbar ist und $\mathfrak{A}$ ein ASM-Zustand ist mit*

- $\mathfrak{A}(pos) =^{\blacktriangledown} s$
- $\sigma \subseteq \mathfrak{A}(store) \circ \mathfrak{A}(env)$

dann erreicht die ASM-Semantik nach endlich vielen Schritten einen Zustand $\mathfrak{B}$ *so, dass*

- $\mathfrak{B}(pos) =^{\blacktriangle} s$
- $\sigma' \subseteq \mathfrak{B}(store) \circ \mathfrak{B}(env)$
- $\mathfrak{A}(env) \subseteq \mathfrak{B}(env)$
- $\mathfrak{A}(frames) = \mathfrak{B}(frames)$

und für alle Adressen $\alpha \in dom(\mathfrak{A}(store)) \backslash ran(\mathfrak{A}(env))$ *gilt*

$$\mathfrak{B}(store)(\alpha) = \mathfrak{A}(store)(\alpha) \text{ und } \alpha \notin ran(\mathfrak{B}(env))$$

12 Compilation auf virtuelle Maschine

12.1 Instruktionen der VM (informale Beschreibung)

`const`(i)	Lege die Konstante i auf den Operandenstack.
`load`(n)	Lade den Wert der $n$ten lokalen Variable auf den Operandenstack.
`loada`(n)	Lade die Adresse der $n$ten lokalen Variable auf den Operandenstack.
`loadi`	Nimm die Adresse zuoberst auf dem Operandenstack und lade den Wert, der an der Adresse gespeichert ist, auf den Operandenstack.
`store`(n)	Speichere den obersten Wert des Operandenstacks als Wert der $n$ten lokalen Variablen.
`storei`	Nimm den obersten Wert des Operandenstacks und die darunterliegende Adresse und speichere den Wert an der Adresse.
`prim`(op)	Nimm die obersten zwei Werte des Operandenstacks und ersetze sie durch das Resultat der Operation op.
`goto`(i)	Springe zur $i$ten Instruktion.
`cond`(op, i)	Teste mittels op den obersten Wert des Operandenstacks. Falls der Test wahr ist, springe zu i.
`invoke`(p, n, k)	Rufe die Prozedur p mit n Argumenten und k lokalen Variablen auf.
`return`	Kehre zurück zum Prozeduraufruf.

12.2 Instruktionen der VM (ASM-Semantik)

$VM =$ **case** $code(\mathsf{pc})$ **of**
`const`$(i) \rightarrow$
  opd:=opd$\cdot [i]$
  pc:=pc+1
`load`$(n) \rightarrow$
  opd:=opd$\cdot$ [mem(loc(n))]
  pc:=pc+1
`loada`$(n) \rightarrow$
  opd:=opd$\cdot$ [loc(n))]
  pc:=pc+1
`loadi`$\rightarrow$
  let $rest \cdot [\alpha] =$ opd **in**
    opd:= $rest \cdot$ [mem(α))]
    pc:=pc+1
`store`$(n) \rightarrow$
  let $rest \cdot [v] =$ opd **in**
    mem(loc(n)):=v
    opd:= $rest$
    pc:=pc+1
`storei`$\rightarrow$
  let $rest \cdot [\alpha, v] =$ opd **in**
    mem(α):=v
    opd:= $rest$
    pc:=pc+1
`prim`$(op) \rightarrow$
  let $rest \cdot [v_1, v_2] =$ opd **in**
    opd:= $rest \cdot [v_1\ \overline{op}\ v_2]$
    pc:=pc+1
`goto`$(i) \rightarrow$ pc $:= i$
`cond`$(op, i) \rightarrow$
  let $rest \cdot [v] =$ opd **in**
    opd:= $rest$
    if $\overline{op}(v)$ **then** pc $:= i$
    else pc:=pc+1
`invoke`$(p, n, k) \rightarrow$
  let $rest \cdot [v_1, \ldots, v_n] =$ opd **in**
    stack:= $push$(stack,$\langle$pc,loc$\rangle$)
    pc:= $first(p)$
    opd:= []
    loc:= $\{1 \mapsto \alpha_1, \ldots, n+k \mapsto \alpha_{n+k}\}$
    forall $i \in [1, \ldots, n]$ **do** mem$(\alpha_i) := v_i$
  where $\alpha_1, \ldots, \alpha_{n+k} =$ **new**(Adr, $n+k$)
`return`$\rightarrow$
  let $\langle pc', loc' \rangle = top($ stack$)$ **in**
    stack:= pop(stack)
    pc:= $pc' + 1$
    loc:= loc'
    opd:= []

12.3 Compilation der arithmetischen Ausdrücke

Funktion: CA : Aexp $\rightarrow$ List(Instr)

Gegeben sei die Anzahl von den lokalen Variablen

und eine Zuordnung von Nummern zu lokalen Variablen und formalen Parametern einer Prozedur.

Falls $x \in \mathsf{Var}$, dann ist $\bar{x} \in \mathsf{VarNr}$.

Für lokale Variablen und Werteparameter x

$\mathrm{CA}(x) =$
  $\texttt{load}(\bar{x})$

Für Variablenparameter x

$\mathrm{CA}(x) =$
  $\texttt{load}(\bar{x})$
  $\texttt{loadi}$

Für Konstanten i

$\mathrm{CA}(i) =$
  $\texttt{const}(i)$

Für binäre Operatoren op

$\mathrm{CA}(e_1\ op\ e_2) =$
  $\mathrm{CA}(e_1)$
  $\mathrm{CA}(e_2)$
  $\texttt{prim}(op)$

12.4 Compilation der Boolschen Ausdrücke

Funktionen $\mathrm{CB}_i : \mathsf{Bexp} \times \mathsf{Pc} \to \mathsf{List(Instr)}$

$\mathrm{CB}_1(b, lab)$: Falls b wahr ist, springe zu lab, sonst ans Ende von b.
$\mathrm{CB}_0(b, lab)$: Falls b falsch ist, springe zu lab, sonst ans Ende von b.

$\mathrm{CB}_1(e_1\ op\ e_2, lab) =$
  $\mathrm{CA}(e_1)$
  $\mathrm{CA}(e_2)$
  $\texttt{prim}(op)$
  $\texttt{cond(ne,}lab\texttt{)}$

$\mathrm{CB}_1(\texttt{not}\ b, lab) =$
  $\mathrm{CB}_0(b, lab)$

$\mathrm{CB}_0(e_1\ op\ e_2, lab) =$
  $\mathrm{CA}(e_1)$
  $\mathrm{CA}(e_2)$
  $\texttt{prim}(op)$
  $\texttt{cond(eq,}lab\texttt{)}$

$\mathrm{CB}_0(\texttt{not}\ b, lab) =$
  $\mathrm{CB}_1(b, lab)$

$\mathrm{CB}_1(b_1\ \texttt{and}\ b_2, lab) =$
  $\mathrm{CB}_0(b_1, end)$
  $\mathrm{CB}_1(b_2, lab)$
$end:$

$\mathrm{CB}_1(b_1\ \texttt{or}\ b_2, lab) =$
  $\mathrm{CB}_1(b_1, lab)$
  $\mathrm{CB}_1(b_2, lab)$

$\mathrm{CB}_0(b_1\ \texttt{and}\ b_2, lab) =$
  $\mathrm{CB}_0(b_1, lab)$
  $\mathrm{CB}_0(b_2, lab)$

$\mathrm{CB}_0(b_1\ \texttt{or}\ b_2, lab) =$
  $\mathrm{CB}_1(b_1, end)$
  $\mathrm{CB}_0(b_2, lab)$
$end:$

12.5 Compilation der Anweisungen

Funktion $\mathrm{CS} : \mathsf{Stm} \to \mathsf{List(Instr)}$

Für lokale Variablen und Werteparameter x

$\mathrm{CS}(x := e) =$
  $\mathrm{CA}(e)$
  $\texttt{store}(\bar{x})$

Für Variablenparameter x

$\mathrm{CS}(x := e) =$
  $\texttt{load}(\bar{x})$
  $\mathrm{CA}(e)$
  $\texttt{storei}$

Leere Anweisung

$\mathrm{CS}\ (\texttt{skip}) = []$

Sequentielle Komposition

$\mathrm{CS}(s_1\ ;\ s_2) =$
  $\mathrm{CS}(s_1)$
  $\mathrm{CS}(s_2)$

If-Anweisung

$\mathrm{CS}(\texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}) =$
    $\mathrm{CB}_0(b, else)$
    $\mathrm{CS}(s_1)$
    $\texttt{goto}(end)$
$else:$ $\mathrm{CS}(s_2)$
$end:$

Lokale Variablendeklaration

$\mathrm{CS}(\texttt{var}\ x := e\ \texttt{in}\ s\ \texttt{end}) =$
  $\mathrm{CA}(e)$
  $\texttt{store}(\bar{x})$
  $\mathrm{CS}(s)$

While-Anweisung

$\mathrm{CS}(\texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}) =$
      $\texttt{goto}(test)$
$whl:$ $\mathrm{CS}(s)$
$test:$ $\mathrm{CB}_1(b, whl)$

Prozeduraufruf (p mit k lokalen Variablen)

$\mathrm{CS}(p(e_1, \ldots, e_m; z_1, \ldots, z_n)) =$
$\mathrm{CA}(e_1)$
$\vdots$
$\mathrm{CA}(e_m)$
$\mathrm{CV}(z_1)$
$\vdots$
$\mathrm{CV}(z_n)$
$\texttt{invoke}(p, m+n, k)$

Für lokale Variablen und Werteparameter z

$\mathrm{CV}(z) =$
$\texttt{loada}(\vec{z})$

Für Variablenparameter z

$\mathrm{CV}(z) =$
$\texttt{load}(\vec{z})$

Prozedurdeklaration

$\mathrm{CS}(\texttt{procedure}\ p(\vec{x}; \vec{y})\ \ \texttt{begin}\ \ s\ \texttt{end}) =$
$\mathrm{CS}(s)$
$\texttt{return}$

13 Denationelle Semantik

13.1 Beschränkte Rekursion

Neue Prozedurnamen $p \restriction n$ für $n \in \mathbb{N}$.

Mit $p \restriction n$ wird die Prozedur p beschränkt auf die maximale Rekursionstiefe n bezeichnet.

- Falls die Ausführung des Prozeduraufrufs $p \restriction n(\vec{e}; \vec{z})$ zu einer Rekursionstiefe grösser als n führt, terminiert der Aufruf nicht.
- Die Prozedur $p \restriction (n+1)$ ruft in ihrem Rumpf nur Prozeduren $q \restriction n$ auf.
- Die Ausführung von $p \restriction 0(\vec{e}; \vec{z})$ terminiert nie.

Definition 13.1. Für eine Anweisung s bezeichnet man mit $s \restriction n$ die Anweisung, die man aus s erhält, indem man alle Prozeduraufrufe $p(\vec{e}; \vec{z})$ durch beschränkte Prozeduraufrufe $p \restriction n(\vec{e}; \vec{z})$ ersetzt.

13.2 Monotonie

Was mit kleiner Rekursionstiefe berechnet werden kann, bekommt man auch mit grösserer Rekursionstiefe.

Lemma 13.1 (Monotonie). *Falls $m \leq n$ und $\sigma \mathcal{R}[s \restriction m]\sigma'$, dann gilt auch $\sigma \mathcal{R}[s \restriction n]\sigma'$*

13.3 Natürliche Semantik ⇒ denationelle Semantik

Theorem 13.1. *Falls in der natürlichen Semantik $\sigma -[s]\rightarrow \sigma'$ herleitbar ist, dann gibt es ein $n \in \mathbb{N}$, so dass in der denationellen Semantik gilt $\sigma \mathcal{R}[s \restriction n]\sigma'$.*

13.4 Denationelle Semantik ⇒ natürliche Semantik

Theorem 13.2. *Falls in der denationellen Semantik $\sigma \mathcal{R}[s \restriction n]\sigma'$ gilt, dann ist in der natürlichen Semantik $\sigma -[s]\rightarrow \sigma'$ herleitbar.*

13.5 Wiederholung: Relationen

Definition 13.2. Eine Relation R auf einer Menge A ist eine Menge von Paaren $R \subseteq A \times A$.

Definition 13.3. Eine Relation R heisst *funktional*, falls für alle x, y, z gilt: Falls $\langle x, y\rangle \in R$ und $\langle x, z\rangle \in R$, dann ist $y = z$.

Komposition und Identität

Zuerst S, dann R.

Definition 13.4. $R \circ S = \{\langle x, z\rangle | \exists y \langle x, y\rangle \in S \text{ und } \langle y, z\rangle \in R\}$

Definition 13.5 (Identität auf A). $Id_A = \{\langle x, x\rangle | x \in A\}$

Die Komposition ist assoziativ und Id_A ist ein Neutralelement.

Lemma 13.2. *Seien R_1, R_2, R_3 und R Relationen auf A. Dann gilt:*

- $R_1 \circ (R_2 \circ R_3) = (R_1 \circ R_2) \circ R_3$
- $Id_A \circ R = R = R \circ Id_A$

Die Komposition von Relationen ist monoton bezüglich der Mengeninklusion.

Lemma 13.3. *Falls $R_1 \subseteq R_2$ und $S_1 \subseteq S_2$, dann ist $R_1 \circ S_1 \subseteq R_2 \circ S_2$*

Die Komposition von Relationen ist distributiv über der Vereinigung von Relationen.

Lemma 13.4. *Sei $(R_i)_{i \in I}$ eine Familie von Relationen. Dann gilt:*

$$(\bigcup_{i \in I} R_i) \circ S = \bigcup_{i \in I} (R_i \circ S), \qquad S \circ (\bigcup_{i \in I} R_i) = \bigcup_{i \in I} (S \circ R_i)$$

Reflexivität und Transitivität

Definition 13.6. Eine Relation R heisst *reflexiv* auf A, falls $\langle x, x\rangle \in R$ für alle $x \in A$.

Definition 13.7. Eine Relation R heisst *transitiv* auf A, falls für alle $x, y, z \in A$ gilt: Falls $\langle x, y\rangle \in R$ und $\langle y, z\rangle \in R$, dann ist $\langle x, z\rangle \in R$.

Lemma 13.5. *Sei R eine Relation. Dann gilt:*

- *R ist reflexiv auf A genau dann, wenn $Id_A \subseteq R$.*
- *R ist transitiv auf A genau dann, wenn $R \circ R \subseteq R$.*

Potenzen und reflexiv, transitiver Abschluss

Die Potenzen R^n und der reflexiv, transitive Abschluss R^*.

Definition 13.8. Sei R eine Relation auf A. Dann definieren wir:

$$R^0 = Id_A, \qquad R^{n+1} = R \circ R^n, \qquad R^* = \bigcup_{n \in \mathbb{N}} R^n$$

Lemma 13.6. *Für alle $m, n \in \mathbb{N}$ ist $R^m \circ R^n = R^{m+n}$*

R^* ist die kleinste reflexiv, transitive Relation die R umfasst.

Lemma 13.7. *Sei R eine Relation auf A. Dann gilt:*

- *R^* ist reflexiv und transitiv auf A und $R \subseteq R^*$.*
- *Falls S eine reflexive und transitive Relation auf A ist mit $R \subseteq S$, dann ist $R^* \subseteq S$.*

Lemma 13.8. *Sei R eine Relation auf A. Dann gilt*

$$R^* = Id_A \cup (R^* \circ R)$$

Funktionale Relationen

Die Komposition von funktionalen Relationen ist funktional.

Lemma 13.9.

- *Falls R und S funktional sind, dann ist $R \circ S$ funktional.*
- *Falls R funktional ist, dann ist R^n funktional.*

Bemerkung. Wenn R funktional ist, folgt nicht dass R^* funktional ist.

13.6 Eigenschaften der denationellen Semantik

Definition 13.9. Für Boolsche Ausdrücke b sei

$$\mathcal{R}[b] = \{\langle \sigma, \sigma \rangle | \sigma \in \mathsf{State}, \mathcal{B}[b]\sigma = 1\}$$

Theorem 13.3. *Für die denotationelle Semantik von Programmen gilt:*

$$\mathcal{R}[\texttt{skip}] = \mathsf{Id}_{State}$$

$$\mathcal{R}[s_1 \ ; \ s_2] = \mathcal{R}[s_2] \circ \mathcal{R}[s_1]$$

$$\mathcal{R}[\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ end}] = (\mathcal{R}[s_1] \circ \mathcal{R}[b]) \cup (\mathcal{R}[s_2] \circ \mathcal{R}[\texttt{not } b])$$

$$\mathcal{R}[\texttt{while } b \texttt{ do } s \texttt{ end}] = \mathcal{R}[\texttt{not } b] \circ (\mathcal{R}[s] \circ \mathcal{R}[b])^*$$

Lemma 13.10.

$$\mathcal{R}[\texttt{if } b \texttt{ then } s \texttt{ end}] = (\mathcal{R}[s] \circ \mathcal{R}[b]) \cup \mathcal{R}[\texttt{not } b]$$

13.7 Die While-Anweisung als kleinster Fixpunkt

Definition 13.10. Sei Γ ein Operator von $\mathcal{P}(M)$ nach $\mathcal{P}(M)$.

- X ist ein *Fixpunkt* von Γ, falls $\Gamma(X) = X$.
- X ist der *kleinste* Fixpunkt von Γ, falls X ein Fixpunkt von Γ ist und für jeden weiteren Fixpunkt Y von Γ gilt, dass $X \subseteq Y$.

Satz 13.1 (Fixpunktsatz). *Für die While-Anweisung gilt:*

$$\mathcal{R}[\texttt{while } b \texttt{ do } s \texttt{ end}] = \text{ kleinster Fixpunkt von } \Gamma_{b,s}$$

wobei der Operator $\Gamma_{b,s}$ von $\mathcal{P}(\mathsf{State} \times \mathsf{State})$ nach $\mathcal{P}(\mathsf{State} \times \mathsf{State})$ definiert ist durch

$$\Gamma_{b,s}(X) = (X \circ \mathcal{R}[s] \circ \mathcal{R}[b]) \cup \mathcal{R}[\texttt{not } b]$$

14 Hoare-Logik

Axiomatische, syntaktische Programmverifikation mit Hoare-Tripel:

$$\{\varphi\} s \{\psi\}$$

Dabei sind φ und ψ logische Formeln: Zusicherungen.

Bedeutung: Falls *vor* der Ausführung der Anweisung s die Formel φ gilt und s terminiert, dann gilt ψ nach der Ausführung von s.

14.1 Das Beweissystem von Hoare

$$\frac{}{\{\varphi\}\texttt{skip}\{\varphi\}} \quad \text{Axiom (}\texttt{skip}\text{)}$$

$$\frac{}{\{\varphi \frac{e}{x}\} x := e \{\varphi\}} \quad \text{Axiom (:=)}$$

$$\frac{\{\varphi\} s_1 \{\psi\} \quad \{\psi\} s_2 \{\chi\}}{\{\varphi\} s_1 \ ; \ s_2 \{\chi\}} \quad \text{Regel (;)}$$

$$\frac{\{\varphi \wedge b\} s_1 \{\psi\} \quad \{\varphi \wedge \neg b\} s_2 \{\psi\}}{\{\varphi\}\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ end}\{\psi\}} \quad \text{Regel (}\texttt{if}\text{)}$$

$$\frac{\{\varphi \wedge b\} s \{\varphi\}}{\{\varphi\}\texttt{while } b \texttt{ do } s \texttt{ end}\{\varphi \wedge \neg b\}} \quad \text{Regel (}\texttt{while}\text{)}$$

$$\frac{\varphi \to \varphi' \quad \{\varphi'\} s \{\psi'\} \quad \psi' \to \psi}{\{\varphi\} s \{\psi\}} \quad \text{Abschwächung}$$

Definition 14.1. Wir schreiben $\vdash \{\varphi\} s \{\psi\}$, falls $\{\varphi\} s \{\psi\}$ in dem erweiterten Beweissystem herleitbar ist.

Ableitbare und zulässige Axiome und Regeln

$$\frac{\{\varphi \wedge b\}s\{\psi\} \quad \varphi \wedge \neg b \to \psi}{\{\varphi\}\texttt{if } b \texttt{ then } s \texttt{ end}\{\psi\}}$$

$$\frac{}{\{\varphi\}s\{\texttt{true}\}}$$

$$\frac{}{\{\texttt{false}\}s\{\varphi\}}$$

$$\frac{}{\{\varphi\}s\{\varphi\}} \quad \text{falls } \mathrm{FV}(\varphi) \cap \mathrm{AV}(s) = \emptyset$$

$$\frac{\{\varphi_1\}s\{\psi\} \quad \{\varphi_2\}s\{\psi\}}{\{\varphi_1 \vee \varphi_2\}s\{\psi\}}$$

$$\frac{\{\varphi\}s\{\psi_1\} \quad \{\varphi\}s\{\psi_2\}}{\{\varphi\}s\{\psi_1 \wedge \psi_2\}}$$

$$\frac{\{\varphi\}s\{\psi\}}{\{\exists x\varphi\}s\{\psi\}} \quad \text{falls } x \notin \mathrm{FV}(s) \cup \mathrm{FV}(\psi)$$

$$\frac{\{\varphi\}s\{\psi\}}{\{\varphi\}s\{\forall x\psi\}} \quad \text{falls } x \notin \mathrm{FV}(s) \cup \mathrm{FV}(\varphi)$$

14.2 Korrektheit der partiellen Hoare-Logik

Definition 14.2. Ein Tripel $\{\varphi\}s\{\psi\}$ ist *wahr* im Sinne der partiellen Korrektheit (geschrieben $\vDash \{\varphi\}s\{\psi\}$), falls für alle Belegungen σ, σ' gilt:

Falls $\mathcal{T}[\varphi]\sigma = 1$ und $\langle \sigma, \sigma' \rangle \in \mathcal{R}[s]$,
dann ist $\mathcal{T}[\psi]\sigma' = 1$.

Falls das Tripel $\{\varphi\}s\{\psi\}$ herleitbar ist im Beweissystem von Hoare, dann ist $\{\varphi\}s\{\psi\}$ wahr.

Theorem 14.1. *Falls* $\vdash \{\varphi\}s\{\psi\}$*, dann* $\vDash \{\varphi\}s\{\psi\}$.

14.3 Totale Korrektheit von Programmen (Terminierung)

Die Regel (`while`) wird ersetzt durch die folgende Regel ($\texttt{while}_{\text{tot}}$):

$$\frac{\begin{array}{l}\{\varphi \wedge b\}s\{\varphi\} \\ \{\varphi \wedge b \wedge t = z\}s\{t < z\} \\ \varphi \to 0 \leq t\end{array}}{\{\varphi\}\texttt{while } b \texttt{ do } s \texttt{ end}\{\varphi \wedge \neg b\}}$$

Dabei muss gelten $z \notin \mathrm{FV}(\varphi) \cup \mathrm{FV}(t) \cup \mathrm{FV}(b) \cup \mathrm{FV}(s)$.
Die Variable z wird benutzt um sich den Wert von t vor Ausführung von s zu merken und ihn mit dem Wert von t nach s zu vergleichen.
Der Term t ist ein Mass, das bei jedem Durchlauf echt abnimmt und so die Terminierung der While-Schlaufe garantiert.
Die Prämisse $\varphi \to 0 \leq t$ stellt sicher, dass das Mass nicht negativ wird.
Der Term t ist eine obere Schranke für die Anzahl Durchläufe der While-Schlaufe.

Definition 14.3. Wir schreiben $\vdash_{\text{tot}} \{\varphi\}s\{\psi\}$, falls $\{\varphi\}s\{\psi\}$ in dem Beweissystem mit neuer Regel für die While-Schlaufe herleitbar ist.

Definition 14.4. Ein Tripel $\{\varphi\}s\{\psi\}$ ist *wahr* im Sinne der totalen Korrektheit (geschrieben: $\vDash_{\text{tot}} \{\varphi\}s\{\psi\}$), falls für alle Belegungen σ mit $\mathrm{FV}(s) \subseteq \mathrm{dom}(\sigma)$ gilt:

Falls $\mathcal{T}[\varphi]\sigma = 1$, dann gibt es ein $\sigma' \in \mathsf{State}$ mit $\langle \sigma, \sigma' \rangle \in \mathcal{R}[s]$ und $\mathcal{T}[\psi]\sigma' = 1$.

$\vDash_{\text{tot}} \{\varphi\}s\{\psi\}$ bedeutet, dass in jedem Anfanszustand, in dem die Vorbedingung φ gilt, die Anweisung s terminiert mit einem Endzustand, in dem die Nachbedingung ψ gilt.

14.4 Korrektheit der totalen Hoare-Logik

Theorem 14.2. *Falls* $\vdash_{tot} \{\varphi\}s\{\psi\}$*, dann* $\vDash_{tot} \{\varphi\}s\{\psi\}$.

14.5 Erweiterung der Hoare-Logik: Arrays

Axiom für die Array-Zuweisung

$$\frac{}{\{\varphi\frac{e_2}{a[e_1]}\}a[e_1] := e_2\{\varphi\}}$$

Problem: Wie definiert man die Substitution $\varphi\frac{e_2}{a[e_1]}$?

Einfacher Fall:

$$\{\underbrace{x < a[j]}_{\varphi\frac{a[j]}{a[j+1]}}\}a[j+1] := a[j]\{\underbrace{x < a[j+1]}_{\varphi}\}$$

Komplizierter Fall:

$$\{\underbrace{x < (k = j+1 ? a[j] : a[k])}_{\varphi\frac{a[j]}{a[j+1]}}\}a[j+1] := a[j]\{\underbrace{x < a[k]}_{\varphi}\}$$

Neue Ausdrücke: $b \ ? \ e_1 \ : \ e_2$ [if b then e_1 else e_2]

15 Dynamische Logik

15.1 Überblick

Modale Operatoren

$[s]$ (Box s) und $\langle x \rangle$ (Diamond s)

Bedeutung der Operatoren

$[s]\varphi$ Falls s terminiert, dann gilt φ nach der Ausführung von s.
$\langle s \rangle\varphi$ Die Anweisung s terminiert und φ gilt nach der Ausführung.

$$[s]\varphi \leftrightarrow \neg\langle s \rangle\neg\varphi$$

16 Bytecode Verification

(Byte)Code Verifikation: Das Programm wird, wenn es geladen wird, statisch analysiert. Nur Programme, die zur Laufzeit keine Checks der defensiven VM verletzen würden, werden vom Verifikator akzeptiert. Solche Programme können dann auf der *normalen* VM ausgeführt werden.

16.1 Verifikationsalgorithmus (Idee)

Die Prozeduren werden einzeln verifiziert.

Der Verifikator führt die Instruktionen symbolisch aus und rechnet nur mit Typen, nicht mit Werten.

Definition 16.1. $\textsc{Vars}(p) = ValueParams(p) \cup VarParams(p) \cup Locals(p)$

Definition 16.2. $\textsc{ValidCodeIndex}(j) \Leftrightarrow 0 \leq j \leq maxCode(p)$

Annahme: Alle Variablenparameter sind Outputparameter, d.h. der Verifikator muss sicherstellen, dass bei einer Return-Instruktion die Variablenparameter sicher einen Wert zugewiesen haben.

16.2 Verifikation (statischer Teil)

Die folgenden statischen Constraints müssen erfüllt sein:

$\texttt{const}(i)$	$i \in \mathbb{Z}$
$\texttt{load}(n)$	$n \in \textsc{Vars}(p)$
$\texttt{loada}(n)$	$n \in \textsc{Vars}(p) \backslash VarParams(p)$
$\texttt{store}(n)$	$n \in \textsc{Vars}(p)$
$\texttt{prim}(op)$	$op \in \mathsf{AddOp} \cup \mathsf{MulOp} \cup \mathsf{RelOp}$
$\texttt{goto}(i)$	$\textsc{ValidCodeIndex}(i)$
$\texttt{cond}(op, i)$	$op \in \{\texttt{eq},\texttt{ne}\}, \textsc{ValidCodeIndex}(i)$
$\texttt{invoke}(q, n, k)$	$n = \lvert ValueParams(q) \cup VarParams(q) \rvert,$ $k = \lvert Locals(q) \rvert$

16.3 Verifikation (dynamischer Teil)

Anfangszustand der Verifikation von p

- $\mathsf{Visited} = \mathsf{Changed} = \{0\}$
- $\mathsf{opdV}_0 = []$
- $\mathsf{locV}_0 = init(p)$

Die Typen der Variablen zu Beginn von p

$$\begin{aligned} init(p) &= \{i \mapsto \mathsf{int} | i \in ValueParams(p)\} \\ &\cup \{i \mapsto \mathsf{adr}(i) | i \in VarParams(p)\} \\ &\cup \{\mathsf{adr}(i) \mapsto \mathsf{undef} | i \in VarParams(p)\} \\ &\cup \{i \mapsto \mathsf{undef} | i \in Locals(p)\} \end{aligned}$$

Verifikationsalgorithmus (top-level Loop)

$\textsc{Verify}=$ **while** $\textsc{Changed} \neq \emptyset$ **do**
  choose $i \in \mathsf{Changed}$ **do**
    if $\textsc{Check}(i, \mathsf{locV}_i, \mathsf{opdV}_i) = \mathsf{True}$ **then**
      $\mathsf{Changed} := \mathsf{Changed} \backslash \{i\}$;
      for all $\langle j, locT, opdT \rangle \in \textsc{Succ}(i, \mathsf{locV}_i, \mathsf{opdV}_i)$ **do**
        $\textsc{Propagate}(j, locT, opdT)$
    else throw `''Check violated''`

Das Propagieren der Typen

$\textsc{Propagate}(j, locT, opdT) =$
  if $j \notin Visited$ **then**
    if $\textsc{ValidCodeIndex}(j)$ **then**
      $\mathsf{locV}_j := locT; \mathsf{opdV}_j := opdT;$
      $\mathsf{Visited} := \mathsf{Visited} \cup \{j\}; \mathsf{Changed} := \mathsf{Changed} \cup \{j\}$
    else throw `''Invalid code index''`
  elseif $locT \sqsubseteq \mathsf{locV}_j$ **and** $\mathsf{opdT} \sqsubseteq \mathsf{opdV}_j$ **then skip**
  elseif $length(opdT) = length(\mathsf{opdV}_j)$ then
    $\mathsf{locV}_j := \mathsf{locV}_j \sqcup locT;$
    $\mathsf{opdV}_j := \mathsf{locV}_j \sqcup opdT;$
    $\mathsf{Changed} := \mathsf{Changed} \cup \{j\}$
  else throw `''Propagation not possible''`

Die Struktur der Verify-Typen

Definition 16.3. Der Typ `undef` ist der grösste Typ:

$$\sigma \sqsubseteq \tau \Leftrightarrow \sigma = \tau \vee \tau = \texttt{undef}$$

Die kleinste untere Schranke $\sigma \sqcup \tau$ von zwei Typen ist `int` oder $\texttt{adr}(n)$, falls beide `int` bzw. $\texttt{adr}(n)$, sonst `undef`.

Vergleich der Typen von Instruktionen

Definition 16.4. Punktweiser Vergleich der Typen der Variablen:

$$locS \sqsubseteq locT \Leftrightarrow \mathrm{dom}(locS) = \mathrm{dom}(locT) \wedge$$
$$\forall i \in \mathrm{dom}(locS)(locS(i)) \sqsubseteq locT(i))$$

Definition 16.5. Punktweiser Vergleich der Typen der Operanden:

$$opdS \sqsubseteq opdT \Leftrightarrow length(opdS) = length(opdT) \wedge$$
$$\forall i \in [1 \ldots length(opdS)](opdS(i) \sqsubseteq opdT(i))$$

Die dynamischen Checks für die Instruktionen

$\textsc{Check}(j, locT, opdT) \Leftrightarrow$ **case** $code(j)$ **of**
  $\texttt{load}(n) \rightarrow locT(n) \neq \texttt{undef}$
  $\texttt{loadi} \rightarrow length(opdT) \geq 1 \wedge \textsc{IsAddr}(top(opdT, 1))$
  $\texttt{store}(n) \rightarrow length(opdT) \geq 1$
  $\texttt{storei} \rightarrow length(opdT) \geq 2 \wedge \textsc{IsAddr}(top(drop(opdT, 1)))$
  $\texttt{prim}(op) \rightarrow length(opdT) \geq 2 \wedge last(opdT, 2) = [\texttt{int},\texttt{int}]$
  $\texttt{Cond}(op, i) \rightarrow length(opdT) \geq 1 \wedge top(opdT, 2) = \texttt{int}$
  $\texttt{invoke}(q, n, k) \rightarrow length(opdT) \geq n \wedge$
    $\forall i \in ValueParams(q)(opdT(i) = \texttt{int}) \wedge$
    $\forall i \in ValueParams(q)\textsc{IsAddr}(opdT(i))$
  $\texttt{return} \rightarrow \forall i \in VarParams(p)(locT(\texttt{adr}(i)) \neq \texttt{undef})$
  otherwise $\rightarrow \mathsf{True}$

Die Berechnung der Nachfolger

$\textsc{Succ}(pc, locT, opdT) = \textbf{case}\ code(pc)\ \texttt{OF}$
$\texttt{const}(i) \to \{\langle pc+1, locT, opdT \cdot [\texttt{int}]\rangle\}$
$\texttt{load}(n) \to \{\langle pc+1, locT, opdT \cdot [locT(n)]\rangle\}$
$\texttt{loada}(n) \to \{\langle pc+1, locT, opdT \cdot [\texttt{adr}(n)]\rangle\}$
$\texttt{loadi} \to \{\langle pc+1, locT, drop(opdT,1) \cdot [locT(top(opdT))]\rangle\}$
$\texttt{store}(n) \to \{\langle pc+1, locT[n \mapsto top(opdT)], drop(opdT,1)\rangle\}$
$\texttt{storei} \to \{\langle pc+1, locT[\alpha \mapsto top(opdT)], drop(opdT,2)\rangle\}$
$\quad \textbf{where}\ \ \alpha = top(drop(opdT,1))$
$\texttt{prim}(op) \to \{\langle pc+1, locT, drop(opdT,2) \cdot [int]\rangle\}$
$\texttt{goto}(i) \to \{\langle i, locT, opdT\rangle\}$
$\texttt{Cond}(op, i) \to \{\langle pc+1, locT, opdT\rangle, \langle i, locT, opdT\rangle\}$
$\texttt{invoke}(q, n, k) \to \{\langle pc+1, locT \oplus out(p, opdT), []\rangle\}$
$\texttt{return} \to \emptyset$

Durch einen Prozeduraufruf mit den Argumenten $opdT$ werden den Variablenargumenten von $opdT$ (= die Adressen auf $opdT$) Werte zugewiesen.

$out(p, opdT) =$
$\{i \mapsto \texttt{int} | \texttt{adr}(i) \in opdT, i \in ValParams(p) \cup Locals(p)\} \cup$
$\{\texttt{adr}(i) \mapsto \texttt{int} | \texttt{adr}(i) \in opdT, i \in VarParams(p)\}$

Das Überschreiben von Bindungen in endlichen Funktionen:

$$f \oplus g = \{(x \mapsto y) \in f | x \notin \mathrm{dom}(g)\} \cup g$$

16.4 Die defensive VM

Bei der defensiven VM tragen die Werte ihren Typ mit sich herum.

Die defensive VM benutzt dieselben Checks wie der Verifikator.

Theorem 16.1. *Falls in einem Programm jede Prozedur eine Typisierung ihres Code-Arrays hat, dann verletzt das Programm keine Checks auf der defensiven VM.*

Konklusion: Falls ein Programm vom Verifikator akzeptiert wird, dann ist es *sicher* auf der normalen VM.